ERDŐS IVÁN

COMMODORE 64 ASSEMBLY

LSI Alkalmazástechnikai
Tanácsadó Szolgálat

COMPUDRUG mfk.

flotisz

A szövetkezet saját fejlesztésű termékcsaládja a FLOTISZ.

Tagjai: floppy fej-tisztító
mágnesszalag egység fej-tisztító
mágneslemezcsomag tisztító
képernyő tisztító és
írógéptisztító készlete.

Hasonló elven alapuló használatuk egyszerű, higiénikus, mint például a

floppy fej-tisztítóké,

amelyek az 5 1/4"-os és 8"-os egységek író-olvasó fejét tisztítják, ezáltal a lemezek élettartamát is megnövelik.
Egy- és kétfejes lemezegységekhez egyaránt használható.

A készlet tartalmaz 1 sprayt és 2 tisztítólemezt.
A sprayvel bepermetezzük a megadott helyen a tisztítólemezt, azt az egységbe helyezzük, és kb. 30 másodpercig olvastatjuk.
Az átlagosan igénybevett gépeknél ezt elegendő havonta egyszer elvégezni.

1 db FLOTISZ készlet 2 évig használható.

Gyártja:

COMPUDRUG mfk.
Műszaki Fejlesztő Kisszövetkezet
1136 Budapest, Fürst Sándor u. 5.
Telefon: 128-128

ERDŐS IVÁN

COMMODORE 64 ASSEMBLY nyelvű programozása

LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT
BUDAPEST, 1985

„A MIKROSZÁMÍTÓGÉPEK ÉS ALKALMAZÁSI RENDSZEREIK KUTATÁSA-FEJLESZTÉSE"

című

OMFB TÁRCAPROGRAM

4/b alprogramjához kapcsolódóan készült

Lektorálta: dr. Úry László

Témafelelős: SZIKLAI KLÁRA

ISBN 963 592 483 6

T A R T A L O M

1.1 Programozási nyelvek mikroszámitógépen

Az olvaso egy olyan könyvet tart a kezében, amelynek célja egy bizonyos mikroszámitógép egy bizonyos programozási nyelvének ismertetése. Mielött azonban szüken vett témánkra térnénk, talán nem felesleges pár szót szólni általában a programozási nyelvekről. A mikroszámitógépekről általában nem érdemes ezen a helyen értekezni, hiszen mindent és mindennek az ellenkezőjét elmondták, leirák már erről a témáról. Szögezzünk le kissé homályosan mindössze csak annyit, hogy **itt és most** mikroszámitogépen a Commodore 64 kategóriáju egyszerü személyi számitógépet értjük.

A programozási nyelvek története nem egyidös a számitógép-pel, hanem annál messzebbre nyulik vissza. Ennek ellenére az elsö számitógépeket még közvetlenül gépikódban, tehát a processzor által közvetlenül megérthetö kódok felhasználásával programozták. A mai értelemben vett programozási nyelvek kialakulása az ötvenes évek második felében kezdödött meg igazán,amikor az elsö magasszintü nyelveket, az ALGOL-t, a COBOL-t, a FORTRAN-t definiálták. Nem kivánunk azonban történelemmel foglalkozni, lássuk a mai helyzetet.

Ma a programozási nyelvek területén az elmult 30 évben lezajlott forradalmi fejlödés ellenére, vagy talán éppen annak következtében a helyzet elég zavaros. Igen sok programozási nyelvet használnak, és ezek néha a legkevésbé sem hasonlitanak egymásra. Mégis próbáljuk meg meghatározni, és egy aránylag elfogadott módon osztályozni öket.

Valamennyi programozási nyelv közös tulajdonsága, hogy számitogépek vezérlésére alkalmas. Ezt a vezérlést nem közvetlenül a gépnek szóló utasitásokkal végzik, hanem olyan nyelvet használnak, amely könnyebben megtanulható, értelmezhetö, átláthato az ember számára, mint a közvetlen gépikód, ugyanakkor az ilyen nyelven irt program egyértelmüen leforditható gépi utasitások sorozatára. Ezt az átalakitást szintén programok végzik, amelynek két alapvetö tipusa ismert, az interpreter vagy értelmezö és a compiler vagy forditoprogram.

Az **interpreter** a programozási nyelv utasitásait az eredeti forráskódban, vagy kissé átkódolt és tömöritett formában sorra veszi és értelmezi, majd azonnal végrehajtja azokat. Ennek lényeges elönyei és hátrányai vannak a forditáshoz képest. Elöny, hogy a program forrásnyelven rendelkezésre áll, bármikor módositható és a módositott program azonnal kipróbálható. A jobb interpreterek még a forrásnyelvü utasitások nyomkövetését is megengedik, a programozó az általa irt utasitások hatását közvetlenül megvizsgálhatja. Lényeges hátránya viszont az interpretereknek viszonylagos lassuságuk. Ha egy utasitást az interpreter egy program futása során egymilliószor hajt végre, akkor egymilliószor értelmezi jelentését,elemzi helyességét és csak azután tudja végrehajtani.

A **forditóprogram** a forrásnyelven irt utasitásoknak közvetlenül végrehajtható gépikódu programot feleltet meg, amely majdnem (!) olyan hatékony, mintha gépikódban irtuk volna. Nagy elönye ennek az eljárásnak, hogy a leforditott program önmagában képes müködni és meglehetösen gyors. Hátránya, hogy módositása a forditási eljárás megismétlését vonja maga után, ami különösen mikroszámitógépen igen idöigényes. Ezenkivül a leforditott program nehezen nyomkövethetö, hiszen az eredeti nyelvi elemek már javarészt nem figyelhetök meg, különithetök el benne.

Mindezekböl következöen az igazán jó nyelvi rendszerek ugyanannak a programozási nyelvnek mind interpreterét, mind forditóprogramját tartalmazzák, söt a programtesztelést is segitik az eredeti nyelvi elemeket használó nyomkövetési eszközökkel. Sajnos ilyen rendszerre a mikroszámitógépek területén alig van példa.

A programozási nyelvek egy lehetséges felosztása aszerint osztályozza a nyelveket, hogy mennyire kell az emberi nyelven megfogalmazott vezérlési utasitást - programot - átdolgozni, hogy az adott nyelven helyes, az eredeti feladatot megoldó programot kapjunk. Eszerint beszélünk:

- assembly nyelvekröl,
- géporientált nyelvekröl (szerencsére erre nemigen van példa mikroszámitógépen),
- magasszintü nyelvekröl (pl. BASIC, FORTRAN, PASCAL, ADA),
- nagyon magasszintü vagy logikai programozási nyelvekröl (pl. PROLOG, LISP, MODULA),
- ujtipusu programozási nyelvekröl (pl. FORTH, C).

A legutolsó csoport meglehetösen furcsán hangzó elnevezése és az uj csoport maga azért keletkezett, mert ezeket a nyelveket nehéz nyugodt lélekkel besorolni az elözö négy osztályba.

Az **assembly nyelvek** (amelynek egy képviselöjével fogunk ebben a könyvben foglalkozni) mindig egy adott processzorcsalád utasitáskészletéhez kötödnek, és csupán a gépi utasitások egyszerübb, könnyen megjegyezhetö irásmódját, valamint nem tul bonyolult kifejezések használatát teszik lehetövé. Az assembly nyelvre (a makrólehetöségektöl eltekintve) jellemzö az egy assembly utasitás - egy gépi utasitás megfeleltetés.

A **géporientált nyelvek** szintén egy hardware-hez kötödnek, de nyelvi elemeik, utasitásaik már több gépikódu utasitásnak felelnek meg. A gépek közel olyan hatékony vezérlését teszik lehetövé, mint az assembly programozással elérhetö.

A **magasszintü nyelvek** legfontosabb jellemzöje, hogy gépfüggetlenek (többé-kevésbé) és aránylag bonyolult feladatok irhatók le egy-egy utasitásukkal. Ezért le kell mondani a csak konkrét hardware-re jellemzö sajátosságok kihasználásáról és a keletkezö (forditott vagy interpretált) gépikód hatékonysága elmarad az assembly programétól. Ezek a nyelvek algoritmikusak, ami azt jelenti, hogy a program a megoldás pontos mikéntjét irja le, ha aránylag magas szintü utasitásokkal is.

A nagyon magas szintü vagy **logikai programozási nyelvek**en - az utóbbi inkább jellemzö megnevezés - irt programok nem a megoldás módszerét, hanem inkább a feladatot magát fogalmazzák meg, azután a feladatot egy, vagy néhány algoritmustipussal oldják meg. Jellemzö példája ennek a PROLOG. Itt a gyors programozhatóságért, az igen jól olvasható programért általában csökkent hatékonysággal kell fizetni. Ugyanekkor ezek a nyelvek alkalmasak olyan bonyolult feladatok megfogalmazására is, amelyeket alacsonyabb szinten emberi korlátokból fakadóan lehetetlen vagy csak igen nagy ráforditásokkal volna lehetséges megoldani. Ilyenek az un. szakértöi rendszerek, amelyeket - néha egymásnak teljesen ellentmondó definiciókkal - ma sokat emlegetnek.

Az **ujtipusu nyelvek** egyrészt a magasszintü nyelvekhez hasonlitanak utasitásaik és programszerkezetük bonyolultságágban, másrészt az assembly nyelvhez az elérhetö gépi lehetöségekben. Tulajdonképpen ezek az igazán géporientált nyelvek, de nem egy bizonyos géptipusra, hanem általában a számitógépekre - legalábbis a mai, un. negyedik generációs gépekre - orientáltak. A mikroszámitógépek esetében igen igéretesek a FORTH nyelv által nyujtott lehetöségek.

A fenti igen rövid eszmefuttatásból is látszik, hogy igazán univerzális nyelvi eszközzel ma még nemigen rendelkezünk. Ezért egy adott feladat megoldásának lényeges eleme a megfelelö nyelv kiválasztása. Ennek során egymásnak ellentmondó szempontok közös optimumát kell megtalálni, hogy az adott gépen rendelkzésre álló nyelvi eszközök közül melyikkel oldjuk meg az adott feladatot aránylag a legkisebb ráforditással és a legnagyobb hatékonysággal.

Ugyanakkor azt látjuk, hogy ma csaknem minden feladat mikroszámitógépes megoldására BASIC-et használnak, mint univerzális eszközt. A BASIC, amely talán a legalacsonyabb szintü "magasszintü" nyelv, valóban aránylag könnyen megtanulható, segitségével gyorsan lehet közepesen bonyolult feladatokat elfogadható módon megoldani. .Ezenkivül egyetlen elönye van, elterjedtsége (igaz, hogy az egyes gépeken használt BASIC nyelvek néha drámaian eltérnek egymástól).

Számos hátránya van. Ahelyett, hogy részletesen elemeznénk [redacted] ezeket, elég itt annyi, hogy mielött elkezdünk egy adott feladatot megoldani, vizsgáljuk meg azért a rendelkezésünkre álló többi lehetöséget is. Persze, akárhogy szidjuk is, a BASIC olyannyira elterjedt, hogy még hosszu ideig a mikroszámitógépek, különösen a hobby-kategória "szabványos" nyelve lesz.

A másik véglet, amivel szintén gyakran találkozunk az, hogy a BASIC nyilvánvalóan nem alkalmas ennek és/vagy bármilyen feladatnak a tisztességes megoldására, ezért a programot assemblyben kell megirni. A következtetés két ponton is hibás. Elöször is nem mindig igaz, hogy a BASIC alkalmatlan az adott és különösen nem, hogy bármilyen feladatra. Másodszor, ha az adott esetben a BASIC alkalmatlan is, még nem biztos, hogy rögtön a legnagyobb ráforditást igénylö eszközzel kell a feladatnak nekiesni. Még az sem igaz,hogy minden feladatot meg lehet oldani assemblyben, mivel egy adott szerkezeti bonyolultság felett az assembly program tökéletes tesztelése szinte lehetetlen. Ha mégis tisztességesen meg akarjuk oldani, annyira szét kell a feladatot kisebb részekre bontani, hogy ezzel szinte uj programozási nyelvet alkotunk, ezáltal vesztünk a hatákonyságból is.

Ezután foglaljuk össze nagyon röviden, hogy milyen programozási nyelveket valósitottak meg a **Commodore 64**-en.

Az assembly nyelvröl nem szólunk itt, mivel ezzel foglalkozik az egész könyv hátralévö része.

A géporientált nyelvek szerencsére elkerülték a Commodore 64-et és nem adtak hozzá egy speciális és másutt alkalmazhatatlan eszközt.

A magasszintü nyelvek közül az elsö természetesen a **BASIC.** A beépitett BASIC interpreterrel nem lehet különösebben dicsekedni, bár amit tud, azt legalább többé-kevésbé hibamentesen tudja.

Bántó hiányosságai a PRINT USING, az ELSE, az ERASE, a tisztességes CHAIN hiánya. A nem kis számu BASIC kiterjesztések (a legismertebbek a SIMON's BASIC és a GRAPHIC BASIC) is sajnos nem a már-már szabványosnak tekinthetö MICROSOFT BASIC, vagy a szintén nem rossz nagygépes BASIC4+ (DEC) irányába terjesztik ki a nyelvet, hanem a Commodore speciális lehetöségeit teszik kényelmesen elérhetövé.

Emlitésre érdemes a C64-en elérhetö nyelvek közül a **PILOT** és **LOGO**, amelyek föleg gyermekek oktatásában hasznosak és a **COMAL**, amelyet a Commodore cég a BASIC helyett ajánl, de nálunk alig ismert.

A többi magas szintü nyelv közül használható formában egyedül a **PASCAL** érhetö el Commodore-n. Itt nem a Data Becker PASCAL 64 forditóprogramjára gondolunk, ami a PASCAL nyelv és általában a forditóprogramirás paródiája, hanem inkább az ügyes ZOOM PASCAL-ra, és méginkább az igazán komoly Oxford-Pascal-ra. Gondot jelent azonban, hogy az ezek által elfogadott nyelvek meglehetösen különböznek egymástól.

A Commodore 64-es **ADA** és **microProlog** forditó ill. interpreter inkább a nyelvek - felületes - bemutatására, mint feladatmegoldásra való.

Az ujtipusu nyelvek közül a **FORTH** megfelelö módon használható interpreterrel - söt több, kisebb-nagyobb mértékben eltéro interpreterrel - rendelkezik Commodore 64-re, bár ezen a gépen való sikereiröl még nem sok hir érkezett.

Ezen rovid bevezetö után, amellyel reméljük nem untattuk az olvasót, térjünk rá könyvünk tárgyára, a Commodore 64 assembly programozásához szükséges ismeretekre.

1.2 Utmutató a 2.-5. fejezethez

Bevezetésül szeretnénk eligazitani az olvasót a további fejezetek felépitésében és rendeltetésében, és néhány tanácsot nyujtani a könyv használatához, valamint egy kicsit foglalkozni az alkalmazott terminológiával.

A 2. fejezet a MOS 6510-es mikroprocesszor müködésével ismertet meg. Igyekeztünk azokkal és csak azokkal az ismeretekkel foglalkozni, amelyek elengedhetetlenül szükségesek az alapszintü assembly programoz elsajátitásához.

A 3. fejezetben a processzor utasitás-készletével foglalkozunk, részletesen tárgyalva valamennyi utasitást.

A 4. fejezet majdnem kézikönyv-szerüen mutatja az assembly programozásban felhasználható legismertebb software-eszközöket. Célunk vele az, hogy az eredeti kézikönyvvel nem rendelkezö vagy annak nyelvét nem értö olvasó az itt leirtak alapján képes legyen ezen eszközök legalább alapfoku szintü alkalmazására. Igyekeztünk a Magyarországon leginkább elterjedt eszközöket összegyüjteni, erényeiket és hibáikat elemezni.

Az 5. fejezet, amely mind közül a legterjedelmesebb, az assembly programozás módszereibe enged bepillantást, és megpróbál enyhiteni a kezdo programozó sulyos gondjain, egyszersmind ötleteket adni a helyes assembly-alkalmazáshoz.

A függelékben néhány - talán hasznos - táblázattal segitjük a gyors tájékozódást egy-egy probléma esetén.

Külön felhivjuk a figyelmet az irodalom-jegyzékre, amely a további - sajnos többnyire nyelvtudást igénylö - tájékozódáshoz igyekszik segitséget nyujtani.

A könyvben tárgyaltak elsajátitását nagymértékben elösegiti a könyvvel együtt, de külön is megvásárolható **lemez**, amely az összes teljes programot és nagyobb lélegzetü szubrutint tartalmazza, természetesen Commodore 64 VC 1541-es formátumban. A lemezen többféle program is található:

- Az /ASS-re végzödö nevüek az assembly nyelvü forrásprogramot tartalmazzák HELP+, azaz BASIC interpreterkód formában. Igy minden segédeszköz nélkül listázhatók, módosithatók.

- Az /MC jelüek az azonos nevü /ASS program gépikódra forditott változatai, amelyek LOAD "<programnév>",8,1 utasitással tölthetök a memóriába. Igy a példákat azok az olvasók is kipróbálhatják, akiknek nincs assemblerük.

- A /BAS névvégzödésü file-ok az assembly programnak megfelelö BASIC programot tartalmazzák - csak a bonyolultabb programok esetén.

- A /TESZT jelü file-ok a megfelelö program tesztelését végzik, segitik elö, szintén nem minden programhoz van ilyen.

- A lemezen lévö egyetlen /TRACE névvégzödésü - BASIC - file olyan speciális /TESZT program, amely a müködést nyomozó változó kiirásaival - reméljük - közelebb visz a bonyolult algoritmus megértéséhez.

Nagyon nehéz egy szakkönyv - s reméljük munkánk annak mondható - pontos olvasótáborát kiválasztani, másként fogalmazva, olyan könyvet irni, amely aránylag széles réteg igényeit elégiti ki. Ez a könyv azzal a szándékkal iródott, hogy a Commodore 64-et ismerö, BASIC programozásban jártas, de nem tul sok assembly ismerettel rendelkezö érdeklödö amatör vagy kezdö hivatásos programozó igényeinek megfeleljen, számukra olyan információkat nyujtson, amelyek segitségével önállóan képesek lesznek egyszerübb assembly programozási feladatok megoldására.

Végezetül a könyv **terminológiájáról.** Napjainkban elég sok vita dul az idegen és különösen a számitástechnikai szakmai kifejezések használatáról, irásmódjáról. Ebben a munkában nem kerültük görcsösen az idegen - jobbára angol - kifejezések használatát, hiszen ezek meglehetösen elterjedtek, ugyanakkor eléggé közismert jelentésüek. Az idegen kifejezéseket - kivéve a teljesen meghonosodottakat - mindenütt az eredeti irásmód szerint irtuk, ami szintén vitatható, de legalább korrekt és következetes eljárás.

2. Mit tud a 6510-es mikroprocesszor. Programozási modell

Ebben a fejezetben azt az eszközt szeretnénk bemutatni, amelynek vezérlése fö feladatunk, a MOS Technology **6510-es jelü mikroprocesszorát.** A 6510-es nem az egyetlen integrált áramkör a Commodore 64-ben, de kétségkivül a legfontosabb, hiszen a központi egység szerepét tölti be, ha a régebbi számitógép-fogalmakkal akarjuk megmagyarázni szerepét. Mint késöbb utalni fogunk rá,s az olvasó bizonyára jól tudja, a gépben (és a perifériákban) található egyéb chipek input/ output vezérlök szerepét töltik be - bár néha igen-igen kiterjesztett értelemben (lásd pl. a VC-1541 lemezegység vezérlöjét,amely egyébként egy MOS 6502-es mikroprocesszor).

A 6510-es mikroprocesszor-chip a MOS Technology korábbi 6502-es gyártmányának kismértékben módositott, továbbfejlesztett változata. Megjegyzendö, hogy a két processzor utasitáskészlete és az utasitások hatása teljesen megegyezik. A 6502-es a **Commodore VIC 20** és **PET** gépének, valamint az **Apple II** sorozatnak lelke többek között. Ezért az ebben a könyvben leirtak a speciálisan Commodore 64-re jellemzö részletek elhagyásával az emlitett gépek programozásához is segitséget nyujthatnak, bár erre a különválasztásra sehol sem utalunk, mivel feladatunk a Commodore 64 programozásának támogatása.

A 6510-es leirásában kizárólag a programozás szempontjából lényeges jellemzöket fogjuk tárgyalni, **kerülve minden**, ebben a tárgyalásban **felesleges hardware-részletet.** Ezért szerepel a fejezet cimében a "programozási modell" kifejezés, mivel itt a mikroprocesszor egy "software kép"-ét adjuk. Ennek során leirjuk azt, hogy mik a 6510-es komponensei és hogyan tartanak kapcsolatot egymással a müködés során.

2.1 Néhány alapfogalom

A 6510-es egy olyan processzor, amely adatokat alakit át a tárolt program utasitásainak engedelmeskedve. Mind az adatok,mind a programutasitások un. gépi memóriában tárolódnak, amelyek fizikailag integrált áramkörökkel vannak megvalósitva a C64-en belül.Ennek a tárolónak két alapvetöen különbözö tipusával kerülünk majd szembe, a **ROM**-mal és a **RAM**-mal. A ROM (Read Only Memory) csak olvasható memória, amelybe a C64 gyártása során információt rögzitettek (többnyire programutasitásokat, bár nemcsak azt). A RAM (Random Access Memory) azaz félrevezetö neve szerint véletlen elérésü tároló a fentivel ellentétben üres, de bármikor átirható, megváltoztatható a tartalma, tetszésünk szerint. Ez a memóriatipus szolgál arra, hogy programjainkat és adatainkat benne tároljuk.

A memória byte-okra, azaz 8 bináris jegyböl álló részekre oszlik, amelynek mindegyike egy külön sorszámmal, cimmel rendelkezik. Egy byte-on 2^8 különbözö számkombináció ábrázolható, amelyeket tetszés szerinti módon értelmezhetünk. Az egyik szokásos értelmezés szerint ezek 0 és 255 közötti számokat jelölnek, egy másik szerint a karakterek (betük, számjegyek, jelek) kódjait, ismét egy másik szerint a különbözö kódok különbözö gépi utasitásokat jelentenek. Hogy egy adott byte-ot, annak tartalmát hogyan kell értelmezni, azt mindig az értelmezönek (a programnak, illetve a programozónak) kell tudni, azt külön semmi sem jelöli.

A 6510-es mikroprocesszor természetesen össze van kapcsolva a **memóriaegységekkel**. Ezt a kapcsolatot **adatbusznak** nevezzük. Ezen a nyolc huzalon keresztül lehetösége van a mikroprocesszornak arra, hogy megváltoztasson egy memóriabyte-ot, illetve kiolvassa tartalmát. Hogy most éppen irni, vagy olvasni kiván-e, azt egy másik adatkapcsolat, a vezérlöbusz egyik bitjével jelzi. A többi bitek egyéb, a külvilággal való kapcsolatokat vezérelnek. A megfelelö byte kiválasztása a memóriából természetesen a cimével (sorszámával) történik. Mivel a feladatok megoldásához meglehetösen nagyszámu programutasitásra és adatra van szükség, a cimek megjelölésére nem 8, hanem 16 bináris jegyböl álló számokat használunk, a megadásukra szolgáló busz 16 párhuzamos elektromos kapcsolatot jelent a központi mikroprocesszor és a memória között. Ilyen módon 2^16 azaz 65536 cim és a neki megfelelö byte különböztethetö, cimezhetö meg. A cimeket és többnyire a memóriabyte-ok tartalmát is hexadecimális számokkal szokás jelölni, amelyeket a 6510-es assembler konvencióinak megfelelöen a $ jellel fogunk mindig bevezetni. Igy egy memóriacim mindig legfeljebb 4-jegyü hexadecimális szám $0000-tól $FFFF-ig, egy byte értéke pedig $00 és $FF közé esik, a határokat is beleértve.

Ebben a munkában nem foglalkozunk olyan elemi ismeretekkel, amelyek a hexadecimális és bináris számábrázolással kapcsolatosak. Tisztázni kivánunk viszont néhány elemi fogalmat az egységes nomenklatura kedvéért. Egy **byte** legbaloldalibb bitje (a byte 7. bitjének is szokás nevezni) a szám elöjelét hordozza, ha az ábrázolt értéket -127 és +127 közötti számként értelmezzük. Ha a 7. bit magas, a számot kettes komplemens ábrázolásu negativ számnak tekintjük. Két egybyte-os szám összeadásával olyan számot kaphatunk, amelynek értéke nagyobb 255-nél, igy egy byte-on már nem ábrázolható. A keletkezett szám jobboldali 8 bitjét nevezzük a müvelet eredményének, mig a legmagasabb helyiértékü bitet átvitelnek (carry-nek).

Ha 255-nél nagyobb számokat kivánunk alkalmazni, természetesen több byte-ot kell együtt kezelnünk. Ennek leggyakoribb esete a **cimek** ábrázolása, amelyre mint tudjuk, a Commodore 64 esetében 16 bitre, **két byte**-ra van szükségünk. A kétbyte-os

egészeket a 6510-es <u>alacsony byte - magas byte</u> sorrendben kezeli a memóriában. Ez azt jelenti, hogy a szám jobboldali két hexadecimális jegye van a kisebb és a baloldali kettö a nagyobb memóriacimen.

Lássunk erre egy kis BASIC nyelvü példát:

Tegyük fel, hogy a memoria $0014-es cimén egy kétbyte-os cim van (tehát a $14 és $15 cimeken, ahol a $14 a kisebb és $15 tartalmazza a nagyobb helyiértékü byte-ot).
A következö BASIC sor ezt a cimet nyomtatja ki decimálisan:

```
KH=PEEK(20):NH=PEEK(21):PRINT 256*NH+KH
```

A 20 és 21 természetesen $14-et, illetve $15-öt jelent, mivel a BASIC nyelvben csak decimális értékek irhatók.

2.2 Regiszterek és jelzök

A 6510-es mikroprocesszor az adatokon bizonyos elöre meghatározott utasitásokat képes végrehajtani. Az **utasitások** egy, két vagy három byte-osak és mindig az elsö byte tartalmazza az utasitásra utaló kódot. Az utasitások végrehajtásához a mikroprocesszor maga is tartalmaz hét byte-nyi adattárolót, amelyeket regisztereknek nevezünk és mindegyikük speciális célokat szolgál. Az utasitások segitségével adatot cserélhetunk a memória és a regiszterek között, valamint logikai és aritmetikai müveleteket végezhetünk ezekkel a tárolt adatokkal. A mikroprocesszor müködése programozási szempontból abból áll, hogy a memóriában tárolt programot utasitásról-utasitásra végrehajtja.

A 6510-es **regiszterei** közül az elsö kétbyte-os, ez a **programszámláló**, röviditve PC (Program Counter). Azért van két byte-ra szükségünk, mert a 6510-es által elérhetö 64 Kbyte-nyi tár egy cimét kell benne tárolni, mégpedig mindig éppen a soron következö utasitás cimét. Hogy melyik a soronkövetkezö utasitás, azt természetesen a mikroprocesszor maga határozza meg a müködés során, attól függöen, hogy éppen milyen utasitást hajt végre. Általában ez az éppen végrehajtott utasitást a memóriában fizikailag követö utasitás, de bizonyos utasitások, vagy külsö állapotok változásai ezt megváltoztathatják, mint a késöbbiekben, az utasitások részletes tárgyalásánál látni fogjuk.

Amikor a Commodore 64 gépet bekapcsoljuk és a processzorra jutó feszültség értéke eléri az 4,75 V-ot,egy un. **inicializálási folyamat** játszódik le, amelynek során a $FFFC-FFFD cimen lévö két byte értéke a programszámlálóba töltödik,azaz ez lesz a bekapcsolás után végrehajtott elsö utasitás cime. Az ilyen, a memóriában tárolt kétbyte-os programcimet vektornak szokás nevezni. A fent leirtakból következik, hogy bekapcsoláskor a memória $FFFC cimén egy programcimnek, tehát értékes információnak kell lenni, ami csak ugy valósitható meg, ha mind ez a memóriacim, mind az, ahova ez a vektor mutat, ROM-ban van.

A következö fontos regiszter az **akkumulátor**, röviditve A. Szoktuk aritmetikai regiszternek is nevezni, mivel a legtöbb aritmetikai (és logikai) utasitásban résztvesz. Ezeknek az utasitásoknak a segitségével az akkumulátorhoz egybyte-os értékeket lehet hozzáadni vagy kivonni belöle és az eredmény mindig az akkumulátorban képzödik. Ezt az uj tartalmat azután a memória tetszöleges cimén tárolhatjuk. Az akkumulátor egybyte-os regiszter.

Az **indexregiszterek** (X és Y regiszterek) alapvetöen két célra szolgálnak, bár ez a két feladat gyakran összeolvad. Egyrészt egyszerü utasitásokkal egyesével növeljük vagy csökkentjük tartalmukat, igy számlálásra használhatjuk öket.

Másik funkciójuk az utasitásokban foglalt memóriacimek módositása, amelyek segitségével bonyolult cimzésmódok valósithatók meg. Ezek a nyolcbites regiszterek szintén a memóriából tölthetök, illetve értékek egy-egy memóriabyte-on tárolható.

A memória második 256 byte-ját (az elsö lapot) tehát a a $0100-tól $01FF-ig terjedö cimtartományt a 6510-es **verem**ként használja. A verem egy speciális tömb, amelyet bizonyos utasitások automatikusan használnak. Itt tárolódnak a szubrutinok, illetve a megszakitások visszatérési cimei. A szubrutinok olyan alprogramok, programrészek, amelyek a program több helyéröl aktivizálhatók, s a hivás helye örzödik meg a veremben, amelynek segitségével talál vissza a program a szubrutinhivást követö utasitásra. A verem soronkövetkezö szabad helyét (a lapon belül) tárolja egy ujabb regiszter, a **veremmutató** angol nevének röviditésével SP (Stack Pointer). A veremmutató kezdeti értéke bekapcsolás után $FF lesz, ami mint cim $01FF-ként értelmezhetö. Mivel a felsö byte mindig $01, ennek tárolására nincs is szükség, igy a veremmutató is egybyte-os regiszter. Az elsö szubrutinhivás (JSR utasitás) hatására a veremben tárolódik a visszatérési cim (két byte-on) és a veremmutató értéke kettövel ($FD-re) csökken. A szubrutin ujabb szubrutint hivhat, ami ujabb cimet ment el a verembe és ez a verem méretéböl következöen legfeljebb 128-szor ismételhetö. Minden szubrutinhivás (logikailag) egy visszatérési (RTS) utasitással kell, hogy végzödjön, amely az utasitásszámlálóba tölti a verem tetején lévö két byte-ot és növeli a veremmutató értékét kettövel, ezáltal folytatja a program végrehajtását a szubrutinhivást követö utasitással.

A szubrutinhiváson kivül az un. **megszakitások** okoznak még automatikus veremmüveleteket. A megszakitások a külsö világból érkezö jelek a mikroprocesszor számára, tehát nem a saját állapotaiból következnek közvetlenül. Két tipusát különböztetjük meg a 6510-es esetében, attól függöen, hogy azok a mikroprocesszor két megszakitásvonalának melyikét aktivizálják, a mikroprocesszor melyik kivezetésének változtatják meg az állapotát. Az egyik, a nem maszkolható megszakitás (NMI - Non Maskable Interrupt) felfüggeszti a program futását, tekintet nélkül arra, hogy milyen állapotban van a processzor. Erre a gép különbözö egységeinek pontos idözitése miatt feltétlenül szükség van. Ezzel szemben a második vonalon jelentkezö megszakitások a program által kizárhatók, maszkolhatók. Ez a maszkolható megszakitások vonala (IRQ - Interrupt Request). Mindkét tipusu megszakitás a verembe menti az utasitásszámláló aktuális értékét (hasonlóképpen, mint a szubrutinhivás) és az állapotregiszter (lásd késöbb) értékét is. Ennek megfelelöen a veremmutató értékét hárommal csökkenti.

Végsö ténykedésként az utasitásszámlálóba tölti a megszakitáskezelö szubrutin cimét tartalmazó vektort. Ez NMI esetén a $FFFA-$FFFB, IRQ esetén a $FFFE-$FFFF memóriacimekröl történik.

Természetesen a megszakitáskezelö szubrutinok is egy speciális utasitással (RTI) érnek véget, amely visszaállitja a megszakitás elötti állapotot. Ezt megelözöen ujra engedélyezi megszakitást a megszakitásjelzö (lásd késöbb) magasra állitásával. Ez hasonlóan müködik, mint a fent leirt RTS utasitás, de az utasitásszámlálón kivül az állapotregisztert is visszamenti a veremböl.

A fentieken kivül külön utasitásokkal is kezelhetjük a vermet, amelyet pl. szubrutinok paraméterátadására használhatunk.

Az utolsó regisztert tulajdonképpen bitenként értelmezzük. Ez az **állapotregiszter**, röviditve P (Program status register) és egyes bitjeinek külön elnevezése is van, ezek a jelzök.

A P regiszter 7. (legbaloldalibb) bitje a **negativ jelzö**, az N. Ennek értékét (mint a többi jelzöét is) bizonyos utasitások megváltoztatják, mások hatástalanok rá. Nevét onnan kapta, hogy értéke magas, ha egy aritmetikai müvelet eredménye negativ, azaz legmagasabb helyiértékü bitje magas. Ezt az értéket feltételes programelágaztatásra használhatjuk.

A V jelzö, vagy **tulcsordulásjelzö** a kettes komplemens tulcsordulást jelzi kivonás esetén. A BIT utasitás szintén kezeli, mint majd ott részletesen tárgyaljuk. Ez a jelzö az állapotregiszter 6. bitje. Az 5. bit értéke közömbös programozási szempontból.

A B vagy software **megszakitás jelzö** az állapotregiszter 4. bitje. Akkor állitódik magasra, ha az éppen végrehajtott utasitás egy BRK (megszakitás) utasitás. Ennek hatása ugyanolyan, mintha egy maszkolható megszakitáskérelem érkezett volna, igy jól használható programok kipróbálása során. Az egyetlen dolog, amiböl a megszakitáskezelö szubrutin megtudhatja, hogy software-megszakitás történt, az a B jelzö. Jó megjegyezni, hogy bár a BRK a maszkolható megszakitást váltja ki, hatása nem maszkolható, ezért hibás (BRK-ra futó) megszakitáskezelö szubrutinok könnyen végtelen megszakitásba torkollhatnak.

A D jelzö, az állapotregiszter 3. bitje a binárisan kódolt **decimális üzemmódot** jelzi, ha értéke 1. Ez az összeadás (ADC) és kivonás (SBC) utasitásra érvényes. Ebben az esetben (decimális módban) egy byte két, négy biten ábrázolt decimális számjegyböl áll, az ábrázolható érték tehát a 0-99 tartományba esik. A decimális mód beállitására és megszüntetésére külön utasitások szolgálnak.

A 2. bit az I, neve **megszakitás maszkolás jelzö.** Ha értéke 1, amit külön utasitással (SEI) érhetünk el, a megszakitások maszkolva lesznek, hatástalanok. Ekkor csak a nem maszkolható (NMI) és software (BRK) megszakitás hajtódik végre. Törlését a CLI utasitás végzi, amely után a program ismét megszakitható.

A Z vagy **nulla jelzö** szerepe az N-éhez hasonló, de nem a negativ, hanem a nulla eredmény állitja magasra. Egyébként az állapotregiszter 1. bitje.

A legkisebb helyiértékü, 0. bit az **átvitel** (carry) **jelzö,** a C. Sok utasitás kezeli, nevét az aritmetikai utasitásokban betöltött szerepéröl nyerte. A feltételes programelágazás gyakori eszköze a Z jelzövel együtt.

A regisztereken kivül a memória elsö két byte-ja is (a $0000 és $0001 cimü) kitüntetett szerepet játszik, tulajdonképpen regiszterként viselkedik. A 0. byte-ot adatirány-regiszternek, az 1.-et input/output kapunak szokás nevezni. A $0001 cimü I/O kapu hat alsó bitje a 6510-es mikroprocesszor-chip egy-egy kivezetésének felel meg. A $0000 cimü adatirány-regiszter egy-egy bitje mondja meg, hogy a neki megfelelö bitet az I/O kapuban olvassa vagy irja a processzor (1=olvasás, 0=irás).Ennek segitségével a hat kivezetésen át a processzor kapcsolatot tud tartani a periférivezérlökkel és egyéb külsö eszközökkel. A perifériális eszközökkel való kapcsolattartás bonyolult folyamat, és nem is csak az I/O kapun át folyik,de vezérlésében a ROM-ban lévö KERNAL szubrutinok is segitenek, igy ez elég egyszerüen végezhetö, mint majd késöbb látni fogjuk.

lásd a 16.-nál a 88-at

2.3 Cimzésmódok

Az egyes utasitások a következö feladatok valamelyikét hajtják végre:

- adatot töltenek a memóriából valamelyik regiszterbe,
- az elözö forditottjaként valamely regiszter tartalmát egy memóriabyte-ba tárolják, vagy más módon módositanak egy memóriabyte-ot,
- egyik regiszterböl egy másikba töltenek értéket,
- megváltoztatják valamely regiszter állapotát,
- megváltoztatják az utasitásszámláló értékét, azaz programelágazást hoznak létre.

Az utasitások elsö byte-ja, mint már emlitettük, mindig az utasitáskód, amely nemcsak magát a feladatot adja meg, hanem a cimzésmódot is, azt hogy a kiegészitö információt (többnyire cimet) hogyan kell meghatározni. A cimzésmód ezenfelül azt is megadja, hogy az utasitáshoz még hány byte információ kapcsolódik, kettö, egy vagy egy sem. A 6510-es mikroprocesszor igen változatos cimzésmódokkal rendelkezik, amelyeket a következökben részletesen tárgyalunk.

Az abszolut cimzéshez hárombyte-os utasitáshossz tartozik, tehát a cim 2 byte-os. Ekkor az utasitáskódot követö két byte a cim maga, kisebb helyiértékü byte - nagyobb helyiértékü byte sorrendben. Ilyenkor tehát az utasitás maga tartalmazza a cimet, amelyet a cim-buszon el kell helyezni.
Ha pl. a soronkövetkezö utasitás igy fest:

```
$AD,$01,$08
```

a $AD utasitáskód azt jelenti, hogy egy byte-ot a memóriából az akkumulátorba kell tölteni (LDA) abszolut cimzésmóddal, tehát a cimet a második és harmadik byte adja meg.
Ezt az utasitást assembly nyelven igy irnánk:

```
LDA $0801
```

Az utasitáskódot az assembly nyelvben a könnyebben megjegyezhetö mnemonikkal (LDA - LoaD Accumulator) helyettesitettük, a cim a jobban olvasható természetes sorrendü, a cimzésmódot nem kell külön jelölni, ez itt abszolut cimzést jelent. Az assembler ezt az utasitást éppen az elözö alakra, a gépi megfelelöre konvertálja.

A nullás lapu cimzés a 6510-es mikroprocesszor azon speciális tulajdonságát használja ki, hogy az a memória elsö 256 byte-ját ($0000-$00FF memóriacimek) kitüntetetten kezeli.
Az utasitáskódok egy külön csoportja nullás lapu cimzést jelöl, ekkor az utasitás csak két byte hosszu, az operandus csak a cim második (kisebb helyiértékü) byte-ját tartalmazza, mivel az elsö byte mindig $00. Igy az utasitás rövidebb lesz és a memória elérése is kevesebb idöt igényel. A legtöbb

assembler a nullás lapu cimzésnél sem kiván külön jelölést, hiszen a cim nagyságából maga is ki tudja számolni, van-e lehetöség a rövidebb cimzésmód alkalmazására. Mint már emlitettük, a nullás lap mellett az egyes lap is kiemelt szerepet játszik, mint verem-memória.

A közvetlen cimzésmód esetén szintén csak két byte-ra van szükség az utasitás megadására. Ebben az esetben a második byte nem cimet, hanem magát az operandust (egy konstans értéket) tartalmazza, ilyenkor nem kell külön a memóriához fordulni a processzornak.

Pl. a

```
LDA #$FF
```

(assembly) utasitás a $FF értéket tölti az akkumulátorba. A közvetlen cimzés assembly jele a #. Az utasitás gépikódu megfelelöje a

```
$A9,$FF,
```

egymást követö két byte-on elhelyezve látjuk, hogy az LDA (Load Accumulator) utasitás kódja közvetlen cimzésmód esetén $A9, mig abszolut cimzésnél $AD volt. Ez, mint már emlitettük, azért van igy, mert az utasitáskód nemcsak az utasitásra magára, hanem a cimzésmódra is utal. Az utasitás és a cimzésmód együtt már egyértelmüen meghatározza az utasitáskódot, és megforditva.

A beleértett cimzésmód esetén az utasitás egyetlen byte-ból, magából az utasitáskódból áll. A legrövidebb alaknak megfelelöen az ilyen cimzémódu utasitások végrehajtási ideje a legrövidebb. Mivel nincs lehetöség operandus megadására, ezek az utasitások mindig csak a 6510-es regisztereivel és jelzöivel és esetleg a veremmel vannak összefüggésben. Egyszerü példa regiszterböl regiszterbe való átvitelek esete, mint pl. a

```
TAX
```

assembly utasitás, amely az akkumulátor tartalmát az X regiszterbe viszi, de vannak bonyolultabb funkcióju utasitások is, mint pl. az

```
RTI
```

megszakitásból való visszatérési utasitás, melynek több feladata is van.

Ebben a könyvben egy kissé eltérünk a 6510-es gépikódu programozását tárgyaló munkák gyakorlatától. Ezekben egy - itt nem használt - további cimzésmódot is tárgyalnak, az un. akkumulátoros cimzést. Az akkumulátoros cimzést a léptetö utasitások (LSR,ASL,ROL,ROR) esetében lehet alkalmazni, ha az operandus az akkumulátor.Mivel ez esetben az utasitás egy

byte hosszu és minden tekintetben megfelel a beleértett cimzésmód jellemzoinek, a továbbiakban akkumulátoros cimzésmod helyett mindig beleértett cimzésmódot fogunk használni. Meg kell ugyanakkor emliteni azt, hogy a fenti utasitások beleértett cimzésmódu változatánál az assemblerek megkivánják az A fiktiv operandus kiirását, (pl. ASL A), amelyet ezért tiltott cimkeként használni.

A relativ cimzésmódot a feltételes vezérlésátado utasitások használják. Ezeknek a segitségével a program bizonyos feltételek teljesülése esetén nem a fizikailag soronkövetkezö, hanem egy, az utasitás operandusa által megcimzett utasitással folytatódik. Ezekröl az utasitásokról a késöbbiekben részletesebben is szólunk, most a cimzésmód bemutatásához ismerjünk meg csak egyet, a BEQ (Branch on Equal) utasitást, amely akkor okoz vezérlésátadást, ha a nulla jelzo magas. Ez általában akkor következik be, ha az akkumulátort nulla értkkel töltöttük, vagy ha egy összehasonlitás eredménye egyenlöségre utal. A következö kis programrészletben az akkumulátorba a nullás lap $14 cimü byte-jának tartalmát töltjük, majd megnöveljük eggyel az Y regiszter tartalmát (INY - Increment Y register), ha a $14 tartalma nem nulla. Az assembly nyelvü utasitássorok elött az utasitás kezdöcime, és az utasitás hexadecimális kódja szerpel byte-onként:

```
C100  A5 14      LDA $14
C102  F0 01      BEQ N
C104  C8         INY
C105  A5 15   N  LDA $15
            .
            .
            .
```

A fenti formátum egyebként az assembler lista alakja.

Ha a $C100 cimen lévö LDA utasitás végrehajtása során a $14-es cim tartalma nulla, az utasitás egyik következménye az, hogy az ST regiszterben lévö nulla-jelzö bit magas lesz. Az ezt követo BEQ utasitás végrehajtása - mivel a feltétel igaz - azt jelenti, hogy az utasitásszámlálóhoz hozzáadódik az operandus (elöjelesen) igy a soronkövetkezö utasitást nem a $C104, hanem a $C104+1=$C105 cimröl fogja venni a mikroprocesszor. Az assembly irásmodban a vezérlésátadás helyét cimkével (N) jeleztük, s a forditás során az assembler gondoskodik a relativ operandus (esetünkben 1) kiszámitásárál. Mivel egy byte értéke, a határokat is beleértve, -128 és +127 közé esik, a legnagyobb relativ ugrás távolsága is csak ekkora lehet, mivel az operandus mindig pontosan egy byteos. Az utasitások részletes tárgyalásánál majd látni fogjuk, mi a teendö, ha messzebbre kivánunk ugrani egy feltételtol függöen.

Hogy a **negativ számok ábrázolását** megértsük, tegyünk egy kis kitéröt. A 6510-es mikroprocesszor (és általában a számitógépek) a negativ számokat un. kettes komplemens kódban áb-

rázolják. Negativ az a szám, amelynek legbaloldalibb bitje (amit elöjelbitnek is szokás nevezni) magas. Most tekintsük az egybyte-os számokat, de az alábbiak igazak a többbyte-os számokra is. Ha egy negativ szám egybyte-os (8 bites) kettes komplemens kódu ábrázolását akarjuk megkapni, a számot hozzá kell adni $100-hoz (decimális 256), az eredmény a keresett ábrázolás lesz. Igy a -1 kettes komplemens kódja $100-1=$FF. Ez a megoldás igen leegyszerüsiti az aritmetikát, hiszen -1+1 számitása

```
   $FF     vagy binárisan    11111111
   $01                              1
  ------------------------------------
  1/$00                    1/00000000
```

A keletkezett szám legfelsö helyiértékü bitjét (amelyet a byte-on már nem is tudunk ábrázolni) a gép a C (carry) jelzöbe viszi, ezt nevezzük átvitelnek. A 8 biten viszont csupa nulla maradt, az eredmény tehát a vártnak megfeleloen eloállt. Legegyszerübb, ha megjegyezzük, hogy az egybyte-os pozitiv számok $00 és $7F közé esnek (0-127), a negativak pedig $FF és $80 közé (-1-töl -128-ig).

Az indirekt cimzésmód egyetlen utasitás, a JMP - feltétlen ugróutasitás esetén használatos. Az utasitás assembly irásmódjában:

```
A579    6C 04 03    JMP ($0304)
```

a zárójel jelzi az assemblernek, hogy a cimzésmód indirekt. Mi a teendö, ha a következö utasitás indirekt ugrás? Az utasitásban szereplö kétbyte-os cim egy memóriabyte-ra mutat. A megjelölt és következö cimen lévö byte kerül a PC-be, azaz ez lesz a következö utasitás cime. A fenti példában tehát a $304-es és $305-ös cimen lévö értékek (pl. $0304 értéke legyen $7C és $0305 értéke $A5) kerülnek a PC-be mint alsó és felsö byte (azaz az utasitás: ugrás a $A57C cimre). Az ilyen kétbyte-os, a memóriába lerakott ugráscimeket vektoroknak szokás nevezni. A BASIC interpreter, mint a késöbbiekben példát fogunk látni rá, éppen a vektorok segitségével teszi lehetövé a felhsználó gépikódu programszegmenseinek beszurását az interpreter feldolgozási folyamatába.

Az indexelt cimzésmódok talán a legbonyolultabbak,ugyanakkor a leghajlékonyabb adatkezelési lehetöséget nyujtják.
Az X és Y regisztereket az indexelt cimzésre való felhasználhatóságukról indexregisztereknek is szokták nevezni. Az indexelés azt jelenti,hogy egy adatbyte cimét nem közvetlenül, hanem két mennyiség összegeként irjuk le. Az elsöt bázisnak, vagy kezdöcimnek, a másodikat pedig indexnek hivjuk, amely megadja, hogy a kezdöcimtöl hányadik byte lett megcimezve.
Ez ugyanaz a mechanizmus, mint amikor egy egyméretü tömb n-edik eleméröl beszélünk, ahol n az elem indexe. A gépikódu utasitás megadja a kezdöcimet és az indexet. A kezdöcim megadható abszolut cimmel, vagy relativ cimzéssel, az index pedig az X vagy Y regiszter valamelyike. Ez a cimzés-

tipus azért nagyon hasznos, mert az indexregiszter értékének megváltoztatásával ugyanaz az utasitás egy másik byte-ot cimez meg, ezt jól ki lehet használni ciklusok szervezésére. Most nézzük az idetartozó három cimzésmódot!

Az abszolut indexelt cimzésmódnál az utasitás egy abszolut cimet - a kezdöcimet - tartalmazza, pl.

```
0400                 STRING=$400
                     .
                     .
2000   A2 0A         LDX     #10
2002   BD 00 04      LDA     STRING,X
```

Az elsö sorban egy cimkét definiálunk az assembler számára, azaz egy szimbólumot,amely a programban a $400 érték helyett fogunk használni. Másszóval a kezdöcim $400. Az LDX utasitás 10-et tölt ($0A) az X regiszterbe. Az LDA utasitásban ",X" utótag jelzi az assemblernek az abszolut indexelt cimzésmódot, ez a gépikódban a $BD utasitáskódban tükrözödik, amely az LDA ilyen cimzésmódu kódja. Hogyan hajtja végre az utasitást a gép? Elöször veszi az abszolut cimet ($400), ehhez hozzáadja az X regiszter tartalmát ($400+$A=$40A) és az igy kiszámitott cimen lévö értéket tölti az akkumulátorba. Ha X értéke az utasitás végrehajtásakor történetesen nem $A, hanem pl. $30, akkor az A-ba a $430 cimü byte tartalma kerül. Az abszolut indexelt cimzésmód mind az X, mind az Y regiszterrel használható, és megvan a nullás lapu megfelelöjük is, ami természetesen csak két byte hosszu utasitást eredményez és gyorsabban is hajtódik végre.

Az indirekt indexelt cimzésmód az indirekt és abszolut indexelt cimzésmódok összeolvasztásaként értelmezhetö. Ez csak nullás lapu cimet tartalmazhat,ennek megfelelöen az utasitás csak 2 byte hosszu, pl.

```
A557   91 22         STA    ($22),Y
```

A ",Y" utótag a zárójellel együtt (ami az indirekcióra utal) jelenti az assembler számára az indirekt indexelt cimzésmódot. A megjelölt nullás lapu cimen ($22) és soronkövetkezö byte-on ($23) egy kétbyte-os cim található, amely az indirekt cimzés kezdöcime. Az ilyen, általában adattömb kezdetére mutató cimet nem vektornak - az a programba mutat -, hanem pointernek (mutatónak) szokás nevezni. Azt mondhatjuk, hogy a $22, $23 egy tömb pointere. Az utasitás végrehajtásának megértéséhez tegyük fel, hogy a $22 tartalma $00 és a $23-é $C0. Ekkor a kezdöcim a $22, mint pointer értéke ($C000) plusz az Y regiszter értéke (pl. $02), azaz példánkban $C002, erre a cimre kerül az A regiszter tartalma. Ez a cimzésmód a leghajlékonyabb az összes között, igen jól alkalmazható.

Az indexelt indirekt cimzésmód elsö pillantásra igen hasonló az elözöhöz, pedig igen fontos, hogy a különbséget lássuk. Ez nemcsak abból áll, hogy a fenti cimzésmód csak az Y, ez pedig csak az X regiszterrel használatos. Az assembly irásmód

```
    2000  A1 20    LDA ($20,X)
```

már jelzi, hogy itt elöbb az indexelést kell végrehajtani és csak azután az indirekciót. Mit jelent ez a gyakorlatban? Tegyük fel, hogy az utasitás végrehajtásakor X értéke 2, mig a memória a következö értékeket tartalmazza:

CIM	ERTEK
$20	$00
$21	$C0
$22	$0A
$23	$C0
.	.

Végrehajtáskor a processzor elöször az utasitás operandusaként megjelölt nullás lapu cimhez ($20) hozzáadja az X értékét (20+$02=$22), majd az ezen pointer által mutatott cimü ($C00A) byte értékét tölti az akkumulátorba. Ez a cimzésmód talán a legritkábban használt a 6510-es assembly programozása során, mivel segitségével általában egy speciális, ritkán szükséges feladat oldható meg (ráadásul azt is meg lehet másképpen oldani). Tegyük fel, hogy van egy pointertömbünk a nullás lapon, amely kétbyte-os pointereket tartalmaz. Az X regiszterrel, mint indexszel ilyenkor ki tudjuk választani a tömb elemét és általa a hivatkozott cimen lévö byte-ot. Ez igen hasznos lenne ugrótáblák kezeléséhez (CASE programstruktura), de szerencsétlen módon ez a cimzésmód vezérlésátadó utasitások esetén nem használható.

2.4 Megszakitások

A megszakitások olyan események, amelyek félbeszakitják a processzor müködését, megváltoztatják bizonyos regiszterek tartalmát és a program futását általában uj pontról`folytatják. A megszakitásokat külsö (pl. periféria) és belsö (órajel, BRK utasitás végrehajtása) tényezök is kiválthatják.Két alapvetö tipusuk van, maszkolható és nem maszkolható megszakitáskérelem. A megszakitás maszkolásán a processzor egy olyan állapotát értjük,amikor a befutó maszkolható megszakitásokat figyelmen kivül hagyja. Erre nagy szükség van pl. a megszakitást kezelö szubrutinban, amelynek müködése során kellemetlen lenne egy ujabb megszakitás, amely ujraindítaná a rutint. Tulajdonképpen van egy harmadik megszakitás is, amellyel már találkoztunk, ez a reset, ami a bekapcsolás esetén való indulást, ill. a tisztázatlan helyzetböl való ujraindulás folyamatát váltja ki. Azért nem szokás ezt megszakitásnak nevezni, mert egyrészt nem`szakit meg semmilyen tevékenységet, másrészt végrehajtása is különbözik a megszakitásokétól, mivel itt a kezdeti állapot elmentése a megszakitott tevékenység zavartalan folytatásához hiányzik.

A megszakitások kezelésének elsö lépése tehát az állapotregiszter (P) és az aktuális cim elmentése a verembe. Ezután az I megfelelö beállitása (ha szükséges) és a megszakitáskezelö rutin cimének betöltése az utasitásszámlálóba következik. Utóbbit a processzor a memória utolsó háromszor két byte-jának valamelyikéböl veszi a következö táblázatnak megfelelöen:

$FFFA	-	$FFFB	Nem maszkolható megszakitás
$FFFC	-	$FFFD	Reset
$FFFE	-	$FFFF	Maszkolható megszakitás

A szubrutinok elöbb-utóbb egy RTI utasitással fejezódnek be, amely visszaállitja a megszakitást megelözö állapotot, azaz visszamenti a veremböl a P és az utasitásszámláló értékét,és folytatja a program futását. A maszkolható megszakitást kezelö programok (és más megszakitásra kényes tevékenységek) tartalmazzák a SEI (Set Interrupt Flag) a megszakitás maszkolására és a CLI (Clear Interrupt Flag) utasitást az ujraengedélyezésre.

A megszakitásokat **általában hardware okok** idézik elö, pl. az órajel, amely a C64-en minden 60-ad másodpercben megszakitást okoz (bár ez programból kikapcsolható).

De lehet megszakitást kelteni **software uton**, azaz program által is, a BRK (Break) utasitás végrehajtásával.
Ez egy szabályos maszkolható megszakitást okoz, az egyetlen különbség a hardware okokból bekövetkezetthez képest, hogy az ST regiszter B jelzöje magasra állitódik. Innen tudható a megszakitást kezelö szubrutinban, hogy a megszakitást BRK utasitás idézte elö. Ez az utasitás nagyon

hasznos a program hibajavitása során, mivel a hibás ágra futó programot a BRK jól követhetöen megállitja. Ennek hasznositására természetesen gondoskodni kell az $FFFE által hivatkozott megszakitáskezelö rutin megfelelö átirásáról. A gép bekapcsolást követö állapotban a BRK BASIC melegindítást hajt végre, hatása megegyezik a RUN/STOP és a RESTORE billentyük egyidejü lenyomásával.

3. Assembly = kényelmes gépikód

3.1 Mit tud egy assembler - mit kellene tudni egy assemblernek

Ha gépi kódu programozásról beszélünk,szinte mindig assembly programozásra gondolunk. Természetesen megvan a mód és lehetöség arra, hogy a gépikódu programot byte-röl byte-ra pl. a POKE BASIC utasitás segitségével elhelyezzük a memóriában és elinditsuk. Kétségkivül ez a legfáradságosabb és a legtöbb hibalehetöséget magában foglaló megoldás. Nem sok ennél kevésbé szórakoztató módja van a számitógépes idötöltésnek, éppen ezért ez a legrosszabb lehetöség, ha a program hossza az 1-2 utasitást meghaladja.

Szerencsére vannak a gépikódu programozásnak olyan eszközei, amelyek kombinálják a gépikód maximális hatékonyságát a gyors, biztonságos és jól nyomonkövethetö munkával és ezek a különféle assemblerek. Az elképzeléseink szerint készült program azonban a Murphynek tulajdonitott aforizma szerint nem vágyainknak, hanem utasitásainknak megfelelöen müködik, ezért a megirt program még alapos csiszolásra szorul. Ebben nyujtanak segitséget a gépikódu monitorok, debuggerek, amelyekröl szintén ejtünk pár szót a késöbbiekben.

Elöbb azonban az **assemblerekröl.** Ezek tulajdonképpen olyan software-eszközök, speciális célprogramok, amelyek a gépikódu programok irására, forditására, dokumentálására valók és alapvetö céljuknak megfelelöen a gépikódu programozás legegyszerübb és egyszersmind leginkább hibamentes módját teszik lehetövé. Sokféle assembler van, amelyek különbözö tipusba sorolhatók, de egy-egy tipuson belül is eltérnek szolgáltatásaikban és az elfogadott nyelv szintaxisát illetöen.

Az assemblerek három fö tipusa

- makróassemblerek,
- szimbólikus assemblerek és az
- azonnali (instant) assemblerek.

Szolgáltatásaik a fenti sorrendben egyre csökkennek. **Az azonnali assemblerek** (jobb gépikódu monitorok tartozéka) egyidejüleg egyetlen gépikódu utasitás forditását végzik, szerepük az utasitásmnemonikok és a cimzésmódok felismerésére és leforditására korlátozódik. A jobb megoldások a relativ cimzésmódnál az ugráshoz szükséges gépikódu operandus kiszámitását is elvégzik, ha operandusként abszolut cimet adunk meg.

A szimbólikus assemblerek már bonyolultabbak. Nem egy utasitást, hanem egy egész programot (vagy programrészt) egyszerre vizsgálnak meg. Az értékekre (cimek, operandusok) szimbólikus megnevezéssel hivatkozhatunk, ami nagymértékben leegyszerüsiti a programok javitását. Általában (több-kevesebb) lehetöséget adnak az adatterületek kijelölésére és kezdeti értékekkel való feltöltésére. Ezt az un. direktivák segitségével teszik, amelyek az utasitáskódhoz hasonló mnemonikok, de nem kerülnek be a leforditott programba, csak a forditást vezérlö, az assemblernek szóló utasitások. Mivel az egyes utasitások leforditásához sokszor olyan információ is szükségeltetik, ami a program szövegében csak később következik, az assembler kénytelen többször is végigolvasni a programot, igy a szimbólikus assembler forditók majdnem mindig több (legalább két-) menetesek.

A makróassemblerek a szimbólikus assembler funkcióin kivül a makródefiniálás és forditás képességeivel is rendelkeznek. A makrok forditási idöben müködö programok, amelyet az assembler interpretál (mint a BASIC interpreter a BASIC kódot) és futásuk eredményeként assembly utasitások keletkeznek, amelyeket az assembler leforditás. Ez a kicsit talán bonyolult kép leegyszerüsitve ugy is jellemezhetö, hogy a makrók kérdöivek, amelyeket a makróparaméterekkel kitöltve helyes assembly utasitásokat nyerünk. Ennek a könyvnek nem célja a makróprogramozás oktatása, bár a témát még érintjük egy makróassembler ismertetése kapcsán.

A következökben részletesen egy szimbólikus assembler példáján ismerjük meg az assemblerek és az assembly programozás lehetöségeit. Az egyértelmü tárgyalásmód kedvéért választanunk kellett egyet a C64-en rendelkezésre álló assemblerek közül, mivel, mint már emlitettük, azok szintaxisa és szolgáltatásai kismértékben eltérnek. A választás a talán leginkább elterjedt, mindenki kezeügyében lévö **HELP+ BASIC bövités** assemblerére esett, amely az alsó középmezönyböl származó, de ezért még használható programtermék.Kezelésére a 4.1 fejezetben visszatérünk, itt csupán azt kivánjuk leszögezni, hogy az ebben a fejezetben szerepeltetett szintaxis (meg a könyvben található összes programlista) a HELP+ assemblerére és csak arra vonatkozik, a többi assemblerre csak több-kevesebb módositással igaz.

3.2 Utasitáskészlet

Az assembly utasitások a következö formájuak:

<sorszám> <cimke> <mnemonik> <operandus> <megjegyzés>

A fenti elemek közül minden esetben kötelezö a <sorszám> és a <mnemonik>, mindig elhagyható a <megjegyzés>, a mnemoniktól függöen kötelezö a <cimke> és az <operandus>. Egy sor csak sorszámból és megjegyzésböl is állhat.

A <sorszám> csak a programok sorainak sorbarendezésére szolgál, tehát a programszerkesztés támogatására, az assembly nyelv szempontjából közömbös.

A <cimke> az utasitás vagy a direktiva azonositására szolgál, az utasitás memóriacimét ill. a direktiva által kijelölt memóriacimet vagy értéket helyettesiti, ahelyett használható. Eppen erröl kapta nevét a szimbolikus assembler.

A <mnemonik> egy gépikódu utasitás hárombetüs emlékeztetö rövidítése vagy egy "."-tal bevezetett direktiva. Valamennyi utasitással és direktivával megismerkedünk a továbbiakban.

Az <operandus> értelmezése a mnemoniktól függ, általában cimet vagy értéket jelöl. Az operandusból következtet az assembler az utasitások cimzésmódjára is, a következöképpen:

<cimkifejezés>	– Utasitáskódtól függöen relativ vagy abszolut. utóbbi esetén, ha a cimkifejezés értéke <$100, nullás lapu abszolut cimzés.
(<cimkifejezés>)	– Indirekt.
(<cimkifejezés>),Y	– Indirekt indexelt.
(<cimkifejezés>,X)	– Indexelt indirekt.
<cimkifejezés>,X és <cimkifejezés>,Y	– Abszolut indexelt.
#<cimkifejezés>	– Közvetlen.

Ha nincs operandus, beleértett cimzémódot használ az assembler (ha az utasitáskód is megengedi). A ROL, ASL, LSR és ROR (ld. késöbb) léptetö utasitások beleértett módu cimzése esetén (amikoris ezek az utasitások az A regiszterre vonatkoznak, az "A" operandust ki kell irni, ami egyébként cimkeként tilos az egyértelmüség miatt. Az operandusban szereplö

<cimkifejezés> szimbólumok (cimkék), decimális, hexadecimális ($-ral kezdödö), bináris (%-kal kezdödö), oktális (§-cal kezdödö) szám, illetve ezek + és - jellel összekapcsolt kifejezése lehet. Speciálisan a <cimke> után (egyes assemblerek esetén elé) irt "<" jel a cim alacsonyabb, ">" jel a cim magasabb helyiértékü byte-ját jelenti. A fentiekre példát a késöbbiekben még böségesen találunk.

A <megjegyzés> ; -vel kezdödö tetszöleges karaktersorozat, amelyet az assembler csak kilistáz, egyébként figyelmen kivül hagy.

Most ismerkedjünk meg végre az utasitásokkal is.

3.2.1. Töltöutasitások

LDA - a <cimkifejezés> által megjelölt byte tartalmát az akkumulátorba tölti
Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N
Példa:

```
LDA CIM
```

LDX - a <cimkifejezés> által megjelölt byte tartalmát az X regiszterbe tölti
Alkalmazható cimzésmódok: KV,N,NY,A,AY
Befolyásolt jelzök: Z,N
Példa:

```
LDX CIM
```

LDY - a <cimkifejezés> által megjelölt byte tartalmát az Y regiszterbe tölti
Alkalmazható cimzésmódok: KV,N,NX,A,AX
Befolyásolt jelzök: Z,N
Példa:

```
LDY CIM
```

3.2.2. Tárolóutasitások

STA - az akkumulátor tartalmát a <cimkifejezés> által megjelölt byte-ba menti
Alkalmazható cimzésmódok: N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: -
Példa:

```
STA CIM
```

STX - az X regiszter tartalmát a <cimkifejezés> által megjelölt byte-ba menti
Alkalmazható cimzésmódok: N,NY,A
Befolyásolt jelzök: -
Példa:

```
STX CIM
```

STY - az Y regiszter tartalmát a <cimkifejezés> által megjelölt byte-ba menti
Alkalmazható cimzésmódok: N,NX,A
Befolyásolt jelzök: -
Példa:

```
STY CIM
```

INC - a <cimkifejezés> által megjelölt byte tartalmát eggyel megnöveli
Alkalmazható cimzésmódok: N,NX,A,AX
Befolyásolt jelzök: Z,N
Példa:

```
INC CIM
```

DEC - a <cimkifejezés> által megjelölt byte tartalmát eggyel csökkenti
Alkalmazható cimzésmódok: N,NX,A,AX
Befolyásolt jelzök: Z,N
Példa:

```
INC CIM
```

3.2.3. Regiszterutasitások

TAX - az A regiszter tartalmát az X regiszterbe tölti
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: Z,N
Példa:

```
TAX
```

TAY - az A regiszter tartalmát az Y regiszterbe tölti
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: Z,N
Példa:

```
TAY
```

TXA - az X regiszter tartalmát az A regiszterbe tölti
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: Z,N
Példa:

```
TXA
```

```
TYA  - az Y regiszter tartalmát  az  A  regiszterbe tölti
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: Z,N
       Példa:
                     TYA

TXS  - az X regiszter tartalmát a veremmutatóba tölti
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: -
       Példa:
                     TXS

TSX  - a veremmutató tartalmát az X regiszterbe tölti
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: -
       Példa:
                     TSX

SEC  - az átvitel-jelzöt magasra állitja
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: C
       Példa:
                     SEC

SED  - a decimális mód-jelzöt magasra állitja
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: D
       Példa:
                     SED

CLC  - az átvitel-jelzöt alacsonyra állitja
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: C
       Példa:
                     CLC

CLD  - a decimális mód-jelzöt alacsonyra állitja
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: D
       Példa:
                     CLD

CLV  - a tulcsordulás-jelzöt alacsonyra állitja
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: V
       Példa:
                     CLV

CLI  - a megszakitás-jelzöt alacsonyra állitja,azaz
       engedelyezi a maszkolható megszakitásokat
       Alkalmazható cimzésmódok: BE
       Befolyásolt jelzök: I
       Példa:
                     CLI
```

SEI - a megszakitás-jelzöt magasra állitja, azaz letiltja a maszkolható megszakitásokat
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: I
Példa:

```
SEI
```

DEX - az X regiszter tartalmát eggyel csökkenti
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: Z,N
Példa:

```
DEX
```

DEY - az Y regiszter tartalmát eggyel csökkenti
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: Z,N
Példa:

```
DEY
```

INX - az X regiszter tartalmát eggyel növeli
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: Z,N
Példa:

```
INX
```

INY - az Y regiszter tartalmát eggyel növeli
Alkalmazható cimzésmodok: BE
Befolyásolt jelzök: Z,N
Példa:

```
INY
```

3.2.4. Vezérlésátadó utasitások

JMP - az utasitásszámláló értékét a <cimkifejezés> által meghatározott cimre állitja, ez lesz a következö végrehajtandó utasitás
Alkalmazható cimzésmódok: A,I
Befolyásolt jelzök: -
Példa:

```
JMP ($304)
```

JSR - szubrutinhivás. Az utasitásszámláló értékét kettövel megnöveli és ezt a cimet a verembe menti. Ezután a programszámlálót a <cimkifejezés> által meghatározott cimre állitja, ez lesz a következö végrehajtandó utasitás. Megjegyzendö, hogy a veremben nem a JSR-t követö uasitás cime van, hanem annál eggyel kisebb érték, de ezzel a programozónak általában nem kell törödnie, mivel ezt a szubrutinból való visszatérési utasitás (RTS) megfelelöen lekezeli.

Alkalmazható cimzésmódok: A
Befolyásolt jelzők: -
Példa:

JSR SETTIM

RTS - szubrutinból való visszatérés. A verem két felső byte-jának eggyel megnövelt értékét az utasitás-számlálóba tölti. Ezzel a soronkövetkező végrehajtandó utasitás a JSR-t követő utasitás lesz a szubrutint hivó progamrészben.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzők: -
Példa:

RTS

RTI - megszakitáskezelő szubrutinból való visszatérés. Az állapotregisztert a verem tetején talált értékkel tölti. A verem soronkövetkező két byte-jának eggyel megnövelt értékét az utasitásszámlálóba tölti.Ezzel a soronkövetkező végrehajtandó utasitás a megszakitást követő utasitás lesz.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzők: valamennyi
Példa:

RTI

BRK - megszakitást okozó utasitás (software megszakitás). A BRK utasitást követő byte cime a verem tetejére kerül, majd az állapotregiszter értéke is kimentődik. Ezután a vezérlés a $FFFE-$FFFF cim által mutatott utasitásra kerül. Letiltja a maszkolható megszakitásokat.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzők: B, I
Példa:

BRK

BCC - az utasitásszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha az átvitel-jelző alacsony.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzők: -
Példa:

BCC CIKLUS

BCS - az utasitásszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha az átvitel-jelző magas.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzők: -
Példa:

BCS CIKLUS

BEQ - az utasitásszámláló tartalmához elöjelhelyesen hozzáadódik az operandus tartalma, ha a nullajelzö magas.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzök: -
Példa:

BEQ CIKLUS

BNE - az utasitásszámláló tartalmához elöjelhelyesen hozzáadódik az operandus tartalma, ha a nullajelzö alacsony.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzök: -
Példa:

BNE CIKLUS

BMI - az utasitásszámláló tartalmához elöjelhelyesen hozzáadódik az operandus tartalma, ha a negativjelzö magas.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzök: -
Példa:

BMI CIKLUS

BPL - az utasitásszámláló tartalmához elöjelhelyesen hozzáadódik az operandus tartalma, ha a negativjelzö alacsony.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzök: -
Példa:

BPL CIKLUS

BVC - az utasitásszámláló tartalmához elöjelhelyesen hozzáadódik az operandus tartalma, ha a tulcsordulásjelzö alacsony.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzök: -
Példa:

BVC CIKLUS

BVS - az utasitásszámláló tartalmához elöjelhelyesen hozzáadódik az operandus tartalma, ha a tulcsordulásjelzö magas.
Alkalmazható cimzésmódok: RE
Befolyásolt jelzök: -
Példa:

BVS CIKLUS

3.2.5. Veremkezelö utasitások

PHA - az akkumulátor tartalmát a verembe menti, a veremmutató által mutatott cimre, majd a veremmutató értékét eggyel csökkenti.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: -
Példa:

PHA

PLA - a veremmutató által hivatkozott cimen lévö értéket az akkumulátorba tölti, majd a veremmutató értékét eggyel megnoveli.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: -
Példa:

PLA

PHP - az állapotregiszter tartalmát a verembe menti, a veremmutató által mutatott cimre, majd a veremmutato értékét eggyel csokkenti.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzok: -
Példa:

PHP

PLP - a veremmutató által hivatkozott cimen lévö értéket az állapotregiszterbe tolti, majd a veremmutato értékét eggyel megnöveli.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: valamennyi
Példa:

PLP

3.2.6. Léptetö utasitások

ASL - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozicióval balra lépteti. A legalsó bit nulla lesz, a kilépö bit az átvitel-jelzöbe kerül. Az utasitás 2-vel való szorzásnak felel meg. A ROL utasitással kombinálva kétbyte-os érték szorzását lehet elvégezni.
Alkalmazható cimzésmódok: BE,N,NX,A,AX
Befolyásolt jelzok: Z, N, C
Példa:

ASL A

vagy

ASL HATVNY

ROL - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozicióval balra lépteti. A legalsó bitpozicióra az átvitel-jelző utasitás előtti értéke lép, a kilépő bit az átvitel-jelzőbe kerül. Az utasitás 2-vel való szorzásnak felel meg. Az ASL utasitással kombinálva kétbyte-os érték szorzását lehet elvégezni.
Alkalmazható cimzésmódok: BE,N,NX,A,AX
Befolyásolt jelzők: Z, N, C
Példa:

```
        ROL A
    vagy
        ROL HATVNY
```

LSR - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozicióval jobbra lépteti. A legfelső bit nulla lesz, a kilépő bit az átvitel-jelzőbe kerül. Az utasitás 2-vel való osztásnak felel meg. A ROR utasitással kombinálva kétbyte-os érték osztását lehet elvégezni.
Alkalmazható cimzésmódok: BE,N,NX,A,AX
Befolyásolt jelzők: Z, N, C
Példa:

```
        ASR A
    vagy
        ASR HATVNY
```

ROR - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozicióval jobbra lépteti. A legfelső bitpozicióra az átvitel-jelző utasitás előtti értéke lép, a kilépo bit az átvitel-jelzőbe kerül. Az utasitás 2-vel való osztásnak felel meg. Az ASR utasitással kombinálva kétbyte-os érték osztását lehet elvégezni.
Alkalmazható cimzésmódok: BE,N,NX,A,AX
Befolyásolt jelzők: Z, N, C
Példa:

```
        ROR A
    vagy
        ROR HATVNY
```

3.2.7. Logikai utasitások

AND - az akkumulátor és a specifikált adatbyte között logikai ES müvelet végzése bitenként. Az eredmény az akkumulátorba kerül. Az eredmény valamely bitje magas, ha mindkét megfelelő bit magas volt, alacsony minden egyéb esetben. Ezáltal egy memóriabyte bizonyos bitjeit lehet az akkumulátorba vinni, mig a többi bit 0 lesz.

Ezt a byte maszkolásának is szokás nevezni.
Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N
Példa:

```
LDA #$7F
AND BYTE
```

ORA - az akkumulátor és a specifikált adatbyte között logikai VAGY muvelet végzése bitenként.Az eredmény az akkumulátorba kerül.Az eredmény egy bitje magas, ha legalább az egyik bit magas volt, alacsony ha mindkettö alacsony volt.Ezáltal két byte tartalmát lehet egyesiteni az akkumulátorban.
Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N
Példa:

```
LDA #$0F
AND BYTE1
STA BYTE1
LDA #$F0
AND BYTE2
ORA BYTE1
```

EOR - az akkumulátor és a specifikált adatbyte között logikai kizáró VAGY müvelet végzése bitenként.
Az eredmény az akkumulátorba kerül.Az eredmény egy bitje magas, ha pontosan az egyik bit magas volt, alacsony minden egyéb esetben. Az utasitás invertálja az egyik byte azon bitjeit,ahol a másik byteban 1 áll.
Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N
Példa:

```
LDA #$FF
EOR BYTE1 ;LOGIKAI KOMPLEMENS-KEPZES
```

BIT - az akkumulátor és a specifikált adatbyte között logikai ES muvelet végzése bitenként. Az akkumulátor értéke nem változik a müvelet végrehajtása során.A nulla jelzö magas lesz, ha az eredmény nulla, a negativ-jelzö a memöriabyte 7., a tulcsordulásjelzö a 6. bit értékét fogja kapni. Az utasitással elönyösen lehet a 6. es 7. bit értékét vizsgálni.
Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N,V
Példa:

```
LDA #$FF
BIT BYTE
BMI BIT7
BVS BIT6
BEQ URES
```

3.2.8. Aritmetikai utasitások

ADC - összeadó utasitás. Az akkumulátor, a megcimzett adatbyte és az átvitel jelzö értékét összeadja és az akkumulátorba viszi.Ha a decimális-jelzö magas, a számolást BCD alakban végzi (ritkán használatos a C64 esetében). Az átvitel-jelzö hozzáadásának a többbyte-os összeadás megvalósitásában van szerepe, ha egyetlen vagy az elsö byte értékét kivánjuk kiszámolni,az utasitás elött az átvitel jelzöt törölni kell.
Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N,V,C
Példa:

```
CLC
ADC MEM
```

SBC - kivonás. Az akkumulátorból kivonja a megcimzett adatbyte-ot és az átvitel jelzö értékének inverzét és az eredményt az akkumulátorba viszi. Ha a decimális-jelzö magas, a számolást BCD alakban végzi (ritkán használatos a C64 esetében). Az átvitel-jelzö levonásának a többbyte-os kivonás megvalósitásában van szerepe, ha egyetlen vagy az elsö byte értékét kivánjuk kiszámolni, az utasitás elött az átvitel jelzöt magasra kell állitani.
Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N,V,C
Példa:

```
SEC
SBC MEM
```

3.2.9. Osszehasonlitó utasitások

CMP - összehasonlitja az akkumulátor és a specifikált adatbyte tartalmát, oly módon, hogy az utóbbit kivonja az elöbbiböl, de az akkumulátor tartalma nem változik, viszont a jelzöket az eredménynek megfelelöen állitja. Ez azt jelenti, hogy az összehasonlitás eredményét az alábbi utasitásokkal lehet vizsgálni:

BPL -> A>=memóriabyte
BMI -> A<memóriabyte
BEQ -> A=memóriabyte
BNE -> A<>memóriabyte

Alkalmazható cimzésmódok: KV,N,NX,A,AX,AY,IX,IY
Befolyásolt jelzök: Z,N
Példa:

```
CMP #$F0
BNE CIM1
```

CPX - összehasonlitja az X regiszter és a specifikált adatbyte tartalmát, olyan módon, hogy utóbbit kivonja elöbbiböl, de az akkumulátor tartalma nem változik, viszont a jelzöket az eredménynek megfelelöen állitja.
Alkalmazható cimzésmódok: KV,N,A
Befolyásolt jelzök: Z,N
Példa:

```
CPX #$F0
BNE CIM1
```

CPY - összehasonlitja az Y regiszter és a specifikált adatbyte tartalmát, olyan módon, hogy utóbbit kivonja elöbbiböl, de az akkumulátor tartalma nem változik, viszont a jelzöket az eredménynek megfelelöen állitja.
Alkalmazható cimzésmódok: KV,N,A
Befolyásolt jelzök: Z,N
Példa:

```
CPY #$F0
BNE CIM1
```

NOP - üres utasitás, teljesen hatástalan.
Alkalmazható cimzésmódok: BE
Befolyásolt jelzök: -
Példa:

```
NOP
```

4. Software eszközök - assemblerek, monitorok, debuggerek

Az assembly programok irásához szerkesztöprogramokat, lefor- ditásukhoz assemblereket használunk. Programszerkesztésre egyes assemblerek esetén a BASIC képernyöszerkesztöjét al- kalmazhatjuk (mintha csak BASIC programot irnánk), más ese- tekben viszont speciális szerkesztöprogramokat. Az assembler is lehet külön program, vagy egy általánosabb fejlesztö rendszer egyik funkciója, mindkettöre láthatunk példát a késöbbiekben.

A programozás természetesen nem ér véget a program leforditásával, bizonyos esetekben ez csak az elsö lépés. Ezután következik a program kipróbálása, tesztelése, javitása egészen addig, mig meg nem gyözödtünk arról, hogy valóban azt a feladatot hajtja végre, amelyre terveztük. Ennek a munkának a során is legalább olyan mértékben rá vagyunk szorulva software-eszközök segitségére, mint a program irása és forditása alatt. Ugyanakkor ennek a tevékenységnek a támogatására igen nehéz kidolgozni általános célu eszközöket, mivel a tesztelendö programok sokkal változatosabb feladatok megoldására lettek tervezve, mint egy átlagos BASIC program. Gondoljuk csak meg, milyen sulyos problémákat vet fel egy gépikódu monitor tesztelése, hiszen nagyon nehéz ugy elemezni müködését, hogy ne avatkozzunk abba bele. Ennek köszönhetöen, különösen a személyi számitógépek területén csak általános és elég egyszerü eszközök terjedtek el. Ezek a **monitorok** és az ezek szolgáltatásain némileg tulmenö **debuggerek** (jobb magyar szó hiján). Mindkettöre látunk példát ebben a fejezetben. Látszólagos egyszerüségük ellenére is igen jó szolgálatot tesznek egy-egy uj program készitésénél. Nagyobb lélegzetü munkák esetén azonban igen lényegessé válik az egyedi, feladatra szabott teszt-ágy és a részletes tesztelési terv elkészitése. Ez nem mindig fedhetö le a debuggerek szolgáltatásaival és bizonyos feladatok esetén magának a problémának a megoldásával azonos mértékü programozási feladatot jelent. Elhagyásával viszont igen rossz minöségü, megbizhatatlan programterméket nyerünk, amellyel több utólagos hibajavitási feladatunk lesz, mint amennyi munkát a tesztelés következetes megoldására kellett volna szánni.

4.1 A HELP+ assembly programozást segitö eszközei

A **HELP+** program elég jól ismert a hazai C64 felhasználók körében. Bár valamennyi területen szolgáltatásai inkább csak átlagosak, nagy elönye, hogy a software-fejlesztés elég széles körét, eszköztárát átfogja. Itt csak az assembly programozáshoz kifejezetten hozzátartozó szolgáltatásait tárgyaljuk, hiszen ez könyvünk témája.

A HELP+ betöltése és elinditása után a BASIC szerkesztöjébe lépünk be, amely lehetövé teszi, hogy a HELP+ assemblere számára forrásprogramot irjunk. Ennek menete teljesen megegyezik a BASIC programok rögzitésével, persze itt szintaktikusan helyes assembly utasitásokat kell irnunk. Ha forrásprogramunk elkészült, azt mentsük ki a BASIC programnál megszokott módon egy lemezfile-ba.

Az **assembly** utasitások irásmódjáról már szóltunk, most ismerjük meg a HELP+ assemblerének direktiváit is. A direktivák az utasitásokkal ellentétben az assemblernek szóló parancsok és a forditás folyamatát vezérlik. Általában a müveleti kód mezöben szerepelnek és pont karakter vezeti be öket.

- **.BYTE** – egy byte tárolóterületet lefoglal és azon elhelyezi a direktiva után álló kifejezés értékét. Ha vesszövel elválasztva több értéket adunk meg, a megfelelö számu byte kerül lefoglalásra. A kifejezés megengedett értéke 0 és 255 között van, a határokat is beleértve.

- **.WORD** – mint a .BYTE, de egy szót, két egymást követö byte-ot foglal le a C64-nél megszokott alacsony helyiértékü byte - magas helyiértékü byte sorrendben.
  Igy pl.

  ```
      CIM   .WORD   $B7F7
  ```

  $F7, $B7 sorrendben lesz lerakva.

- **.TEXT** – a direktivát aposztrófok vagy idézöjelek között követö string karaktereinek ASCII kódjai lesznek lerakva az aktuális cimtöl kezdve.

- **.DISP** – mint .TEXT, de 6 bites ASCII kódok kerülnek lerakásra (azaz C64 screen-kódok).

- **.END** – a forrásprogramot záró direktiva, ha nem teszszük ki, az assembler a forráskód után generálja 65535-ös látszólagos sorszámmal.

- **.LOAD** – a direktiva után irt nevü programmal folytatódik a forditás folyamata. Az assembler "kedves" hibája folytán a név hosszának pontosan meg kell egyeznie az éppen forditás alatt álló programéval.

= - (egyenlöségjel) a szimbólum értékadás direktivája. Az elötte (a cimke mezöben) álló szimbólum az utána (az operandus mezöben) álló kifejezés értékét veszi fel. Mind elötte, mind utána szerepelhet a "*" szimbólum, amely a forditás aktuális cimét tartalmazza, azaz azt a memóriacimet, ahova a soronkövetkezö byte-ot el fogja helyezni az assembler. Néhány példa:

*=$COOO

A következö utasitás vagy adat a $COOO-ra kerül.

ALFA=*

ALFA az elhelyezés-mutató aktuális értékét veszi fel.

```
TEXT    .TXT  'ABCDE'
THOSSZ=*-TEXT
```

THOSSZ a szöveg hosszának értékét kapja.

=+20

Ez is szabályos. 20 byte lefoglalását jelenti.

Az assembler elinditása a sor elejére irt "**[**" karakterrel kezdödik, ezután az assembler három kérdést tesz fel:

- az elsö kérdésre a lemezfile nevével kell válaszolni, amely a forrásprogramot tartalmazza,
- a második kérdés ("Hexa korr. poke:") lehetoséget ad arra, hogy a leforditott gépikódu program ne a saját helyére, hanem az itt megadott értékkel kisebb (!) cimre kerüljön. Erre pl. akkor van szükség, ha a program tervezett helye a HELP+ által lefoglalt $8000-töl kezdödö 16 Kb-ba esik. Ha itt "RETURN"-t adunk, a program a forrásban elöirt helyre kerül,
- a harmadik kérdésre egy háromjegyü bináris számmal kell válaszolni, amely a listázásra utal. Az elsö jegy helyén álló 1-essel hibajelzéseket, a második pozición állóval forráslistát, a harmadikkal a szimbólumtábla kinyomtatását válthatjuk ki. Utóbbi a cimkék neve és a hozzájuk rendelt érték hexadecimálisan. Bekapcsolt nyomtató esetén a forditási lista ott is megjelenik.

Az assembler felhasználja a BASIC területet ($801-töl) a szimbólumtábla tárolásához, igy a memóriában lévö program törlödik.

Sajnos a leforditott program lemezrementéséhez vajmi kevés segitséget nyujt a HELP+. A forditás eredménye a tárba kerül, a forrásprogramban meghatározott helyre. A forditási listából (és csak abból) meghatározható a program elsö és utolsó utasitásának a cime, azaz a program kezdete és vége. Ha a memória $2B-$2C cimére beirjuk a program elsö bytejának a cimét, mig a $2D-$2E cimére az utolsó byte-énál

eggyel nagyobb cimet (a .END direktiva mellé irt cim a listában) a BASIC programterület kezdetét és végét jelzö pointereket állitottuk át. Az ezt követö SAVE BASIC utasitás éppen leforditott programunkat viszi lemezre. A pointerek átállitását akár POKE BASIC utasitással, akár a HELP+ monitor funkciójával végezhetjük. Utóbbival egyszerübb, mivel ebben az esetben a hexadecimális értéket nem kell decimálisra átszámolni. A program kimentése után ne felejtsük el a $2B-$2C és $2D-$2E mutatókat egyaránt $801-re, a BASIC terület elejére állitani, majd egy NEW-t végrehajtani, mert egyébként nem kivánt módon átirhatjuk a memóriát. Ehelyett egy SYS 64738 utasitással szintén alaphelyzetbe hozhatjuk a BASIC interpretert.

A következökben foglalkozzunk az általunk ismertetésre kerülö programtesztelési eszközök legegyszerübbikével, a HELP+ monitor funkcióival. Ezek sajnos igen szegényesek, ezért rájuk is igaz, ami a HELP+ assemblerére, csak igen kisméretü programok készitésénél nyujtanak értékelhetö segitséget.

A **monitorba** a "]" karakter sor elején való kiadásával léphetünk be. Ekkor rögtön megjelenik a képernyön a legutoljára kezelt memóriabyte cime és értéke hexadecimális szám formájában. Ezen aktuális cim értékét a "+" karakter után beirt (mindig hexadecimálisan értendö) értékkel változtathatjuk meg. A "RETURN" megnyomásával a következö byte-ra ugrik a monitor. A memória tartalmát a "/" megnyomása után irt értékkel megváltoztathatjuk. A "RETURN"-nel a következö byte-ra átterve módositási módban maradunk egészen egy ujabb (pl. "+") parancs kiadásáig. Legyünk nagyon óvatosak a memóriamódositás kipróbálása során! Az elsö két lap bizonyos cimeinek és persze a HELP+ által elfoglalt terület ($8000-$A000) megváltoztatása kellemetlen következményekkel járhat. A monitoron belül a "[" kiadásával a nyomtatóra, a "]" megnyomásával a képernyöre irányithatjuk az outputot. Hasonlóan kapcsoló szerepet játszanak a "<" és ">" jelek. "<" kiadásával a géphez csatlakoztatott VC 1541-es lemezegységének memóriájára kapcsolunk át, ezután minden müvelet arra és nem a C64 központi tárára vonatkozik egészen a ">" utasitás kiadásáig, ami helyreállitja az eredeti állapotot. A "*" karakterrel az aktuális cimen kezdödö gépikódu programra adódik a vezérlés. Megjegyzendö, hogy ez a parancs lemezegység-módban (tehát "<" után) is müködik, de akkor a lemezegység processzorára és tárolójára vonatkozik. Kiadásával természetesen óvatosan kell bánni.

A monitor maradék funkciói a disassemblerre vonatkoznak. Ennek segitségével a memória tartalmát assembly formára emlékeztetö listává lehet visszairni. A visszairhatóság természetesen korlátozott, nem vontkozhat a cimkére és megjegyzésre, hiszen ezeknek semmi nyomuk a gépikódu programban. A "-" paranccsal az aktuális cimtöl kezdödö egyetlen utasitást disassemblálhatunk. A szóköz billentyü megnyomásával az aktuális cimtöl kezdödö folyamatos visszaforditást indithatunk meg, amely egészen a "RUN/STOP" megnyomásáig tart. Hosszabb programrész visszaforditását természe-

tesen célszerü nyomtatóra végezni, igy utólag kényelmesebben elemezhetjük az eredményt. A C64 és a VC 1541 ROM-jának tanulmányozásához azonban csak végsö esetben ajánljuk ezt a módszert, mivel ebben a vonatkozásban igen jó könyvek állnak rendelkzésre, ha nem is magyar nyelven (ld. függelék).

A HELP+ monitorából a BASIC-hez az "=" paranccsal térhetünk vissza.

4.2 Makróassembler fejlesztö rendszer C64-re

A Commodore Business Machines által forgalmazott fejlesztö rendszer leirását inkább szemléltetönek, eligazitásnak szánjuk, semmint kézikönyvnek és különösen nem tankönyvnek, mivel a teljes leirás által megkivánt terjedelem messze meghaladja azt, amit e könyv keretében rászánhatunk. Igaz ez a 4. fejezet soronkövetkezö alpontjaira is.

A makróassembler fejlesztö rendszer már sokkal komolyabb programfejlesztö eszköz az elözöekben megismert HELP+-nál. Az assembler forditóprogram makrók használatát is megengedi, gazdagabb hibajelzéssel rendelkezik, használata kényelmesebb. A DOS-szal, a forrásnyelvi szerkesztövel, a keresztreferencia készitövel, a két töltöprogrammal és a monitorokkal együtt kerek rendszert képez, jól használható nagyobb lélegzetü assembler programok készitése során is.
Bár - mint a késöbbiekben látni fogjuk - léteznek jobb, az assembly programozást jobban támogató eszközök is,igen elterjedt a C64 programozók körében.

A programirás és forditás eszközeit a "BOOT ALL" programmal tölthetjük be. Ezután az "EDITOR64" program vezérlése alá kerulünk. A látszat az, mintha szokványos BASIC rendszerben lennénk, de ez nem igaz, mert az EDITOR64 alatt a programsorok interpreter-kódba váltása nem történik meg, ezért az ekkor beirt BASIC programok hibásak lesznek. Az EDITOR64 tehát csak assembly nyelvü programok bevitelére és javitására alkalmas. A program szerkesztésében használhatjuk a BASIC-képernyoszerkesztö minden lehetöségét és még néhány igen praktikus kiegészitését. Ezek dióhéjban a kovetkezok:

AUTO - automatikus sorszámprompt.
Az AUTO n mindaddig sorszámpromptot ad az uj sor elejére n novekménnyel, mig az AUTO (operandus nélkul) parancsot ki nem adjuk.

NUMBER - sorok ujraszámozása.
A NUMBER n1, n2, n3 parancs ujraszámozza n1-töl n2 kezdöértékkel n3 növekménnyel a program sorait.

GET - töltés lemezröl.
A GET "név", n1, n2, n3 parancs a "név" nevü lemezes file tartalmával az n1 sorszámu, már a memóriában lévö sortól kezdve felülirja a programok tárolására szolgáló területet. n2 és n3 az egység- és csatornaszám, alapértelmezés szerint 8 (ha nem adjuk meg). Az n1 is elhagyható, de ha megadjuk, a parancs felhasználható forrásfile-ok egymás után füzésére is.

PUT - mentés lemezre.
A PUT "név", n1-n4, n2, n3 parancsban n2 és n3 jelentése, mint GET-nél. Ha n1-et és esetleg n2-t is megadjuk, a parancs csak a két sorszám közé esö programrészt menti a "név" nevü fileba.

CPUT - tömöritett mentés.
paraméterei és használata megyegyezik a PUT-éval, de a file-t tömöritve, az összes felesleges space elhagyásával menti.

FIND - karaktersorozat keresése.
A FIND /str/, n1-n2 utasitás az str karakterlánc valamennyi elöfordulását kilistázza a teljes programban, vagy n1 és és/vagy megadása esetén a jelzett sortartományban. A "/" karakter helyett tetszöleges, a stringben nem szereplö delimiter használható.

CHANGE - szövegcsere.
A CHANGE /str1/str2/, n1-n2 ugy müködik, mint a FIND, de a megadott sorszámtartományban (vagy a teljes programban) az str1 valamennyi elöfordulását str2-re cseréli.

DELETE - sorok törlése.
A DELETE n1-n2 parancs (szintaxisa mint a LIST-é) a megadott sorszámtartomány sorait törli a memóriából.

FORMAT - formázott lista.
A FORMAT n1-n2 parancs a listázást formázva végzi, azaz külön oszlopban kezdi a cimeket, utasitás-mnemonikokat, operandusokat és megjegyzéseket az assembler listához hasonlóan. Kicsit buta, igy cimkének tekint minden, az elsö pozición kezdödö szót és mnemoniknak minden sor elsö szavát, ha nem az elsö oszlopban kezdödik.

KILL - kikapcsolja az EDITOR64-et, visszatér a BASIC-editorba. Az editor egyébként a SYS 49152 BASIC paranccsal inditható ujra.

Az assembly program megirása során lényegében ugyanazt a formátumot kell alkalmazni, mint a HELP+ esetén az alábbi eltérésekkel. A kifejezésekben a + és - jelek mellett a * és a /, azaz szorzás és osztás is használható. Az operandusmezöt követö megjegyzések elött nem kötelezö kitenni a pontosvesszöt, de ajánlatos az egységesség kedvéért, hiszen nem minden utasitásnak van operandusa. A cim alsó ill. felsö byte-jára a szimbólum elé irt < vagy > jellel utalhatunk. Az operandus nemcsak szám, ill. számtani kifejezés lehet, hanem

egy "'" (aposztróf) jellel megelözött karakter is, ekkor az operandus értéke a karakter ASCII kódja. A direktivák köre szélesebb, mint a HELP+-nál. Az ugyanolyan jelentésü .BYTE, .WORD, .END és az "=" jel mellett a következök használhatók:

.DBYTE - ugyanugy 2 byte elhelyezésére szolgál, mint a .WORD, de a byte-okat magas helyiértékü byte - alacsony helyiértékü byte sorrendben rakja le, eltéröen a 6510-es konvenciótól.

.PAGE - lapdobást okoz az assembler-listában. A direktiva után aposztrófok közé irt szöveg a direktivát követö minden lap tetejére kiiródik,amig egy ujabb szöveggel ellátott .PAGE nem jön.

.SKIP - az utána álló kifejezés értékének megfelelö -ha hiányzik, egyetlen- sort emel az assembler listában. Sem a .SKIP, sem a .PAGE nem jelenik meg a listán.

.OPT - a forditási opciók beállitására szolgál a forditás közben. A lehetséges opciók:

ERRORS - NOERRORS - hibafile keletkezése
LIST - NOLIST - listázás
GENERATE- NOGENERATE - a stringek kinyomtatása

Elegendö az opciók elsö 3 karakterét megadni.

.FIL - hatásában megegyezik a HELP+ .LOAD direktivájával, azaz a forditás a direktiva után álló nevü lemezes file tartalmával folytatódik.

.LIB - ugy müködik, mint a .FIL, de a könyvtári file végigolvasása után az assembler folytatja a félbehagyott file forditását. A .LIB-bel aktivizált file nem tartalmazhat .LIB-et, de .FIL-t igen. Nagyon hatékony eszköz szubrutinkönyvtárak létrehozására!

A maradék direktivák a makródefinicióval kapcsolatosak. Ezeket egy kicsit részletesebben tárgyaljuk és eközben megpróbálunk bepillantani a makróprogramozás rejtelmeibe, ami jól alkalmazva igencsak megnöveli programozói hatékonyságunkat.

A makróprogramozás tulajdonképpen makródefiniálásokból és makróhivásokból áll. Ezek az elemek szabadon keverhetök az assembly sorokkal. A makró-definició a következö szerkezetü:

```
<cimke>      .MAC     <makrónév>
        <utasitások>
<cimke>      .MND
```

A cimkék elhagyhatók, egyébként a makró által generált első és utolsó utasitáshoz tartoznak. Az <utasitások> tetszőleges szöveget tartalmazhatnak,kivéve a .MAC és .MND direktivákat. A makróhoz legfeljebb 9 db paraméter tartozhat. A formális paramétereket a makró-definicióban ?1,?2 ...?9-cel jelöljük. Ezek ugynevezett pozicionális paraméterek, azaz az ?1 utal a makróhivásban szereplö elsö, a ?2 a második stb. aktuális paraméterre. A paraméterek szövegtipusuak, azaz a formális paraméterek a makródefinicióban a makróhivásban szereplö aktuális paraméterek szövegével helyettesitödnek, minden formai ellenörzés nélkül. Ha az assembler egy

<makrónév> <paraméterek>

alaku makróhivást talál a programban, ahol a <paraméterek> szövegek vesszövel elválasztott sorozata,a következöt teszi:

- sorra veszi a makródefinició sorait,
- egy adott sorban minden formális paramétert szövegszerüen helyettesit a hivásban sorszámának megfelelö aktuális paraméter-stringgel. Ha nem adtunk meg egy, a makródefinicióban hivatkozott paramétert,az assemler egy Lxxx alaku paraméter-stringet generál, ahol xxx 001-töl folyamatosan növekvö szám.Ez biztositja, hogy a generált paraméterek minden hiváskor különböznek egymástól.
- ha minden behelyettesitést elvégzett, az assembler ugy tekinti a sort, mintha az a forrásfile-ból jött volna, azaz leforditja.
- a makródefinició utasitásai között természetesen a behelyettesitések után elöállhat makróhivás is, az ilyen hivások azonban legfeljebb 8-as mélységig ágyazhatók egymásba. A makró hivhatja önmagát is.

Ez persze igy egy kicsit tul tömény, lássunk néhány példát. Ezek a példák tulajdonképpen az 5. fejezetbe illenének, amely a programozás technológiájával foglalkozik, tekintsük öket egy kis elörelapozásnak. Az assembly nyelvü programozás során különösen gyakran fordul elö, hogy ugyanazokat az utasitásokat irjuk le, egymástól alig-alig különbözö elemekkel. Az ilyen utasitás-sorozatok az assembly programozás elemi épitökövei, amelyek szinte makróba kivánkoznak. Pl. két egybyte-os szám összeadása elött törölnünk kell az átvitel-jelzöt (carry), ha helyes eredményt akarunk kapni. Eszerint a C=A+B összeadást

```
LDA A
CLC
ADC B
STA C
```

módon irhatjuk le.

Ha a feladat történetesen X=Z+22, az utasitássorozat:

```
        LDA Z
        CLC
        ADC #22
        STA X
```

A hasonlóság szembeszökö. Sokkal egyszerübb, ha irunk egy makrót:

```
        .MAC  OSSZ1B
        LDA   ?1
        CLC
        ADC   ?2
        STA   ?3
        .MND
```

alakban, amelyet ha a következöképpen hivunk:

```
        OSSZ1B  A,B,C
```

éppen az elsö, ha igy:

```
        OSSZ1B  Z,#22,X
```

a második példát kapjuk a makró kifejtésekor.

Egy másik, kicsit összetettebb illusztráció az igen gyakran szükséges ciklusszervezést könnyiti meg:

```
        .MAC  CFEJ
        LDX   ?2
   ?1K  CPX   ?3
        BEQ   ?4
        BPL   ?1V
  ?4    .MND

        .MAC  CVEG
        INX
        JMP   ?1K
    ?1V .MND
```

A két makróból álló együttes egy alkalmazása:

```
        LDA     #0
        CFEJ    C1,#1,#10
        STA     ALFA,X
        CVEG    C1
```

A makróprogram kifejtése:

```
          LDA   #0
          LDX   #1
C1K       CPX   #10
          BEQ   L001
          BPL   C1V
L001      STA   ALFA,X
          INX
          JMP   C1K
C1V
```

A CFEJ makrónak három paramétere van, a ciklus neve - ebből képezzük a ciklusfej és a ciklusvég cimkéit -, a ciklusváltozó kezdőértéke és a maximális ciklusváltozó érték, amelyre még le kell futtatni a ciklust. A makrók természetesen feltételezik, hogy az X regisztert, amit ciklusváltozóként alkalmazunk, a ciklustörzs utasitásai nem módositják. Az alkalmazási példa egy memóriatömb nullázása (ALFA+1-től ALFA+10-ig). A CFEJ makró hivásakor hiányzó negyedik paraméter (?4) helyére az assembler egy cimkét generált, mégpedig olyat, amely a CFEJ egy másik hivásában generálttól feltétlenül különbözik, evvel biztositja a cimkék egyedi voltát. A ciklusvég természetesen ugyanazt az első paramétert kapja, igy válik lehetövé a visszaugrás a ciklus elejére, a generált C1K cimkére. Megjegyzendö még, hogy a ciklus felsö ciklusvég-vizsgálatot alkalmaz, ha tehát belépéskor a ciklusvég-feltétel nem elégül ki (a ciklusváltozó kezdőértéke nagyobb mint a végérték) a ciklus egyszer sem fut le.

Reméljük a fenti példák felkeltették az olvasó érdeklödését a makróprogramozás iránt. Mielött azonban továbbhaladnánk a makróassembler fejlesztöi rendszer megismerésében, nem szabad elhallgatnunk azt, hogy az ismertetett makróassembler talán minden makróassemblerek legszegényesebb példája. A kicsit bonyolultabb feladatok megoldásához rettenetesen hiányoznak a makróváltozók (lokális és globális egyaránt). Ugyanugy szegényes a makrókönyvtár kialakitási lehetöség is (egyedül a .LIB direktiva áll rendelkezésünkre), bár sok nagygépes makróassembler sem nyujt többet! Vegyük viszont figyelembe a programtermék megitélésénél, hogy más assembler készitök a makrólehetöséggel még csak nem is foglalkoznak.

Az editor segitségével megirt és lemezre mentett assembly programot a következökben leirt módon fordithatjuk le. Elöször töltsük be az ASSEMBLER64 programot és RUN-nal inditsuk el. A forditás során az assembler két file-t hoz létre, a betöltö programok számára készitett tárgyprogram-file-t és a cimkehivatkozásokat tartalmazó file-t, amelyből a **CROSSREF64** programmal készithetünk keresztreferencia-listát. Az assembler néhány kérdéssel indul, amelyek (és az

elvárt információ):

OBJECT FILE	–	leforditott program file neve Ha a név üres,nem készül OBJECT file, csak ellenörzés és/vagy keresztreferencia
HARD COPY	–	sornyomtató lista kell vagy nem kell
CROSS REFERENCE	–	keresztreferencia-file készül-jön-e. Megjegyzés: Ha a program tartalmaz .LIB direktivát, az OBJECT file és a keresztreferencia egyszerre nem készithetö el!
SOURCE FILE NAME	–	forrás file neve

Ha az assembler futását meg akarjuk szakitani, hogy a képernyön megjelenö listát megvizsgáljuk, ezt a RUN/STOP billentyü megnyomásával érhetjük el. A továbbinditás bármilyen billentyüvel lehetséges, kivéve a B-t, amely befejezi az assembler futását és visszatér a BASIC operációs rendszerhez.

Az assembler által elöállitott tárgyprogramot a **két töltöprogram** valamelyikével tölthetjük a memóriába (vigyázat, a tárgyprogram még nem végrehajtható program!).
A két program muködésében azonos, de más helyre töltödik a memóriába ($800-ra ill. $C800-ra), igy megkonnyiti a felhasználói program tetszöleges helyre töltését.
Betöltés (LOAD "töltönév",8,1) után a **LOLOAD64** RUN, a **HILOAD64** SYS 51200 utasitással inditható. A töltöprogramok elfogadnak egy eltolási cim (offset) értéket is, igy lehetövé válik, hogy a programokat az eredeti cim + eltolási cim helyre töltsük be. A töltöprogramok lefutása után a felhasználói program futásra kész.
A töltöprogramokhoz hasonlóan a fejlesztöi rendszernek **két monitorprogramja** is van, amelyek szintén csak a betöltési cimeikben különböznek egymástól, igy lehetövé téve a memória tetszöleges helyén lévö (nem tul hosszu) program vizsgálatát. Az XVM4.8 a hexadecimális $8000, az XVM4.C a $C000 cimre töltödik be és SYS 8*4096 ill. SYS 12*4096 utasitással inditható. A parancsok a gépikódu monitornál szokásosak, de a szintaxis megismerése végett röviden összefoglaljuk azokat. A cimek mindenütt négyjegyü hexadecimális számokat jelölnek a bevezetö $-jel elhagyásával.
Az egyetlen kivétel az azonnali assembler nullás lapu cimzése, amit kétjegyü hexadecimális cim jelez.

A parancsok:

R – A regiszterek tartalmának kiiratása. A képernyöre kiirja PC, P, A, X, Y regisztereket és a veremmutatót. A kiirt sor elején lévö ";" tulajdonképpen módositó parancs, ezért ha a sort megváltoztatjuk és RETURN-t nyomunk meg, a regiszterek értéke módosul.

M - Memóriatartalom kiiratása. A parancs utáni elsö a kezdöcim, az elhagyható második a végcim. A kiirás hexadecimális formáju, soronként 8 byte kerül listázásra. A kiirt sorok elején lévö ">" itt is módosító parancsként müködik, mint R-nél.

I - Memória kiiratás karakteresen is. Mint az M parancs, de a byte-ok karakter-megfelelöjét is kiirja soronként.

D - Visszaforditás (disassemblálás) a megadott tartományban. A disassembler által kilistázott utasitások megfelelnek az azonnali assembler szintaxisának, ennek megfelelöen megvan a lehetöség a képernyöszerkesztö segitségével az utasitások helyben történö módositására. Vigyázat: Az utasitások hoszszabbra cseréjével a mögötte álló utasitások átiródnak, jelentésük megváltozik! Rövidebb utasitásra való csere esetén a felszabadult byte-okat töltsük fel gondosan NOP utasitásokkal, hasonló meggondolásból!
A listázó (M, I, D) parancsok által teleirt képernyö a cursor-mozgató billentyü használatával felle görditheto a parancsban megadott határokon tul is (kár részletesebben elmagyarázni, tessék kipróbálni, nagyon hasznos lehetöség).

A - Azonnali assembler. Utasitás-szintaxisa megegyezik az assemblerével, de természetesen cimkék, direktivák, makrók nem használhatók. Igen rövid, néhány utasitásos programok kipróbálására, illetve programjavitásra tanácsos csak használni.

G - Program inditás. A mögötte álló cimre adja a vezérlést. Ha a cim elmarad, a programszámláló akruális értékétöl indul tovább (R parancs!).

F - Memória feltöltés. Az elsö cimtöl a másodikig a harmadik paraméterként megadott értékkel tölt fel minden byte-ot. Pl. az

F C000 C100 00

parancs $C000-tól $C100-ig nullázza a memóriát.

T - Memóriaterület áthelyezése. A paraméterek jelentése:
elsö - az átmásolandó terület kezdöcime
második - az átmásolandó terület végcime
harmadik - a fogadó terület kezdöcime

N - Program áthelyezése. Az elsö két paraméter egy memóriaterületet jelöl ki, amelyen a parancs hatásos lesz. A harmadik paraméter értéke (offset) hozzáadódik minden hárombyte-os (abszolut és in-

direkt cimzésmódu) utasitás operandusához ugy, hogy a 16 bitröl esetleg tulcsorduló eredmény utolsó 16 bitje iródik csak be természetesen a memóriába. Az utasitás hatása korlátozható a negyedik és ötödik paraméterrel, amelyek azt az átkódolás elötti cimtartományt jelölik ki, ahova az átkódolandó operandusok mutattak. A parancs jól kiegésziti a T parancs hatását, ha programról van szó. Mindazonáltal igen könnyü vele programot tönkretenni, különösen, ha a programutasitások között adatok is vannak, mint általában. Ilyenkor néha egész meglepö módon átalakulhat a program. Ha az assembly forráskód is rendelkezésünkre áll, különösen hoszszabb program esetén, tanácsosabb a programot ujraforditani a megfelelö helyre.

H - Byte-sorozat keresése. Az elsö és a második paraméterek által meghatározott cimek tartományában megkeresi a harmadik, negyedik, stb. byte-ok egymást követö elöfordulásait és ezek kezdöcimeit kiirja. A byte-füzér aposztróffal bevezetett ASCII karakterláncként is megadható. Pl. a

H 0801 0900 'ALFA

parancs az "ALFA" karaktersorozat elöfordulásainak kezdöcimét adja meg $801 és $900 között.

A fennmaradó két parancs a programok töltését és mentését végzi:

L - Az elsö paraméter (idézöjelek közötti string) által megjelölt programfile-t a második paraméter által meghatározott perifériáról (01 - kazetta; 08, 09 - lemez) az eredeti helyére tölti.

S - Az elsö két paraméter jelentése megegyezik L-ével. A mentés a harmadik és a negyedik paraméter által megadott cimtartományra vonatkozik. A negyedik paraméter az elsö olyan byte cime, ami már nem lesz kimentve (végcim+1).

Az elözöekben megismert fejlesztöi rendszer mikrogépes rendszerek átlagos, ugyszólván klasszikus eszközének tekinthetö, amely alkalmas a komoly fejlesztö tevékenység támogatására, de a makrólehetöségen kivül egyéb csemegével nemigen büszkélkedhet.

4.3 PROFIMAT

A PROFIMAT, amelyet az NSZK-beli **DATA BECKER** cég forgalmaz, szintén rendszernek mondható, bár mindössze két, lazán kapcsolódó programterméket tartalmaz, a PROFI-MON monitorprogramot és a PROFI-ASS assemblert. A monitor kisértetiesen hasonlit a 4.2 fejezetben ismertetett fejlesztöi rendszer monitorjára, ugyhogy csak a különbségeket irjuk le pár szóban.

A **monitort** a LOAD "PROFI-MON 64",8,1 paranccsal tölthetjük be és SYS 12*4096-tal aktivizálhatjuk. Hiányzik az A és az I parancs, utóbbi funkcióját az M teljesiti. Az A, vagyis az azonnali assembler hiánya bántó hiba. Egyebekben minden parancs és azok szintaxisa megegyezik az elözö fejezetben megismert monitoréval.

Egy kicsivel több figyelmet érdemel az **assembler**, ha nem is különös erényei, de elterjedtsége miatt. Szabályos szimbólikus assemblerröl van szó, néhány eredeti, bár nem tul hasznos bövitéssel.

Az **utasitások szintaxisa** szinte megegyezik az eddig ismertetett assemlerekével. A kifejezések lehetöségei igen kibövültek, a következö müveleti jelek alkalmazhatók:

\+ összeadás,
\- kivonás,
* szorzás,
! logikai vagy müvelet,
& logikai és müvelet,
^ logikai kizáró vagy müvelet,
\> nagyobb helyiértékü byte (egyoperandusos müvelet)
< kisebb helyiértékü byte (egyoperandusos müvelet).

Az egybyte-os értékek negativ számként is irhatók (-1-töl -128-ig), ezek kettes komplemens kódu alakjukkal helyettesitödnek.

A **direktivák** köre is kibövült a már más assemblereknél megismertekhez képest. Változatlan jelentéssel használható a

.BYTE
.WORD
.FILE
.END

direktiva. Erdekes programozási lehetöségeket biztosit az

.IF
.GOTO

direktivák alkalmazása. Segitségükkel feltételes forditást végezhetünk, pl. elöre megtervezhetjük a tesztelést szolgáló részek későbbi kihagyását a programból. Az .IF direktiva után logikai kifejezés, majd az egyébként .IF nélkül is használható .GOTO következik. A .GOTO után tetszöleges programbeli sorszám állhat.

Mielött tovább folytatnánk a direktivák ismertetését, foglalkoznunk kell a PROFI-ASS egy érdekes sajátosságával, amihez további direktivák kapcsolódnak. A PROFI-ASS tervezöi igen élénk forditásideji tevékenységet képzeltek el és igyekeznek támogatni. Bár itt nincsenek makró-lehetöségek, mégis nagyon fontos, hogy élesen megkülönböztessük a forditásideji tevékenységeket a leforditott program futása közben lejátszódóktól, még akkor is, ha ez - mint jelen esetben - erösen összefonódik. A PROFIMAT assemblere felkészült arra, hogy hivását (ami egy BASIC SYS utasitás) programból adjuk ki és forrásszövegét is ugyanott a "BASIC" programban találja. Ezáltal a BASIC nyelvet, mint egy nagygépes operációs rendszer munkavezérlö (job-control) nyelvét használva elöre megtervezhetjük a forditás, tesztelés munkáját.
A programban talált assembly utasitásokat leforditja az assmbler és a nekik megfelelö gépikódot közvetlenül a memóriába tölti a HELP+-hoz hasonlóan, a direktivákat azonnal végrehajtja, BASIC utasitásra találva pedig visszaadja a vezérlést a BASIC interpreternek. Nézzük, milyen direktivák segitik elö ezt a müködésmódot:

<- - Az assembly programban egy szimbólum csak egyszer szerepelhet a cimke mezöben, ezzel értéket kap és a következökben ezzel az értékkel helyettesitödik minden elöfordulása. A <- szimbólum (értékadás) lehetöséget ad arra, hogy a szimbólumnak nem ilyen egyszeri, hanem szabadon változtatható értéket adjunk s ezáltal a forditás vezérlésére használhassuk fel. A hatás nagyon hasonló a makróassemblernél hiányolt makróváltozók használatához.
Egy példa:

```
 80  CIM1=$33C
 90  CIM2=$C000
100  MUTATO<-0
110  LDA  CIM1+MUTATO
120  STA  CIM2+MUTATO
130  MUTATO<-MUTATO+5
140  .IF MUTATO<16 :.GOTO 110
```

A fenti programrészlet forditásideji ciklust szervez. A futó programban hatása lehet, hogy hasonló egy futásideji cikluséhoz, de gyökeresen másképp valósitja azt meg. Megfelel a hagyományosan (.IF nélkül és változó nélkül) irt követ-

kezö programnak, vagyis ugyanazt a tárgykódot eredményezi:

```
 80 CIM1=$33C
 90 CIM2=$C000
100 LDA  CIM1
110 STA  CIM2
120 LDA  CIM1+5
130 STA  CIM2+5
140 LDA  CIM1+10
150 STA  CIM2+10
160 LDA  CIM1+15
170 STA  CIM2+15
```

Ugyanezt a feladatot futásideji ciklussal hasonlóképpen (de nem pontosan ugyanugy és más tárgykóddal) a következö programrészlettel oldhatjuk meg:

```
 80             CIM1=$33C
 90             CIM2=$C000
100             LDX  #0
110 CIKLUS      LDA  CIM1,X
120             STA  CIM2,X
130             TXA
140             CLC
150             ADC  #5
160             CMP  #16
170             BMI  VEGE
180             TAX
190             JMP  CIKLUS
200 VEGE
```

A példák értelmezését és összehasonlitását az olvasóra bizzuk.

A következö direktivák a szimbólumtábla kezelését végzik. A szimbólumtábla az assembler által felépitett táblázat, amely a szimbólumokat és a hozzárendelt értékeket tartalmazza.

.SST <egységszám>,<másodlagos cim>,"<filenév>"
A teljes szimbólumtáblát a megadott egységre, általában egy szekvenciális lemezfile-ba menti, késöbbi felhasználás céljából.

. LST <egységcim>,<másodlagos cim>,"<filenév>"
Az elözö direktiva ellentéte, a korábban .SST-vel kimentett szimbólumtáblát a memóriába tölti.

A két direktiva **alkalmazásával lehetöség** nyilik nagy programok **részleteinek** külön **forditására**, de az egyes **részek** között, a kapcsolat megteremtésére (hasonlóan a szerkesztöprogramok – linkage editorok – müködéséhez). A korábban forditott programban szereplö szimbólumokra hivatkozhatunk a később forditott programban, ha az elsö forditás során a szimbólumtáblát kimentjük, a második forditás elsö lépéseként pedig visszatöltjük.

Végül ismerkedjünk meg az .OPT (opciók) direktiva lehetséges paramétereivel. A direktiva után vesszövel elválasztva akárhány paraméter állhat.

P – Hatására az assembler formázott forráslistát késziт a képernyön. Ha nincs érvényben, akkor is listázásra kerülnek a hibás sorok és a .FILE direktivát tartalmazó sorok.

P<filesorszám>
– Hatására a lista az assembler elinditása elött OPEN utasitással megnyitott file-ra – leggyakrabban printerre – kerül.

O – A tárgykód egy speciális memóriaterületre – a BASIC tömbök helyére – kerül, s onnan tetszés szerint kimenthetö

OO – A tárgykód az eredeti helyére kerül, a programban magában megadott tárolóterületre. Ha ez a terület valamilyen okból foglalt, az elözö módszert (O) kell használni.

O<filesorszám>
– A tárgykód az assembler elinditása elött OPEN utasitással megnyitott file-ra kerül. A kazettás egység itt nem alkalmazható.

N – Az utána felsorolt paraméterek hatását törli. Ha egymagában áll, valamennyi opciót hatástalanitja.

Az **assembler használatáról** is szólunk néhány szót!
Az assemblert a LOAD "PROFI-ASS 64",8,1 utasitással tölthetjük be lemezröl, majd az akár programból, akár közvetlen utasitásként kiadott SYS 9*4096-tal indithatjuk el.
Az assembler a BASIC terület végén helyezkedik el, mintegy 34 Kbyte szabad területet hagyva a program és a szimbólumtábla számára.
Egy tipikus PROFIMAT program:

```
 10  OPEN 5,8,5,"TARGYKOD":REM EBBE A FILEBA FOG FORDITANI
 20  OPEN 3,4:REM NYOMTATO MEGNYITASA
 30  SYS 9*4096 :REM ASS HIVASA
 40  .OPT O5,P3; TARGYKOD 5-OS, LISTA 1-ES FILERA
 50  .LST 8,8,"KONYVTAR"; SZIMBOLUMTABLA TOLTESE
```

```
 60   *=FOPRV; FORDITAS A FORPROGRAM UTANRA
 70   KOD1=$C1; KODTABLA
 80   KOD2=$C9
 90   ELLEN     LDA  PARAM
100             CMP  #KOD1
110             BEQ  K1
120             CMP  #KOD2
130             BEQ  K2
140   HIBAS     LDA  #$FF
150             RTS
160   K1        LDA  #1
170             RTS
180   K2        LDA  #2
190             RTS
200   ELLENV    =*
210   .SST      8,8," :KONYVTAR"
220   .END
230   CLOSE  3:CLOSE  5: PRINT "VEGE A FORDITASNAK"
```

Néhány dolgot érdemes megfigyelni a fenti programban. Az 50-es sorban lévö direktivával betöltjük a korábban kimentett szimbólumtáblát a memóriába. Ezáltal lehetöségünk nyilik a jelen programban nem - de feltételezhetöen a korábban kimentett szimbólumtáblában - szereplö PARAM és FOPRV szimbólumokra hivatkozni. A 210-es sorban a fenti KONYVTAR nevü file-t felülirjuk a jelen program forditása során kiegészitett szimbólumtáblával, ezáltal egy következö forditás már az itt megadott szimbólumokat (pl. a szubrutin kezdöcimét) használhatja.

Osszefoglalva a PROFIMAT által nyujtott szolgáltatásokat (amelyek közul a legbonyolultabbakat és legnehezebben felhasználhatókat nem is tárgyaltuk) megállapithatjuk, hogy érdekes lehetöségeket felvonultató fejlesztöi rendszerröl van szó, amely azonban nem éppen a strukturált programozás és mérnöki tevékenység tipusu szervezett programfejlesztés támogatására szolgál, inkább az érdeklödö amatörök számára készült.

4.4 SM-KIT 64

Az SM-KIT 64 lényegében a HELP+-hoz hasonló, de annál valamivel szélesebb lehetöségeket biztositó alapsoftware. Három részböl áll: a BASIC programozást és programtesztelést, a gépikódu programozást és végül a VC 1541-es lemezegység processzorának programozását támogató részekböl. A következökben természetesen csak a témánkat, az assembly programozást érintö részletekröl szólunk. Erdekes módon, mig a rendszer az eddig ismertetett eszközöknél valamivel hatékonyabban támogatja a programok tesztelését, kipróbálását, az assembly programok forditását szinte egyáltalán nem, mivel mindössze egy debuggerhez illö azonnali assemblert tartalmaz. Természetesen egy tetszöleges assembler által leforditott program már kezelhetö az SM-KIT segitségével.

Az SM-KIT 64 három része egyetlen programot képez, amelyet a programlemezröl a LOAD "*",8,1 utasitással tölthetünk a memóriába. Ezután az SM-KIT "."-tal kezdödö parancsai használhatók, amelyeket az alábbiakban ismertetünk.

.<cim> - disassembler funkcio, amely egy képernyönyi outpotot ad, majd RETURN-re ujabbat. RUN/STOP-pal leállitható. A disassemblálás kezdöcimét a <cim> adja meg.

.<cim><assembly utasitás>
- lehetöséget ad egyetlen utasitás leforditására a <cim> által jelölt memóriaterületre. A cimek megadása alapállapotban decimálisan történik. Ez hexadecimálisra változtatható .H megadásával, ami decimálisra változtathato .$-ral. A szám után irt H vagy $ egyszerüen hexadecimális - decimális ill. decimális - hexadecimális átváltást tesz lehetövé a kijelzési mód megtartásával.

.<cim>B
- A <cim>-töl kezdve decimális módban 5, hexadecimálisban 8 byte-ot ir ki. RETURN megnyomásával ujabb sor kerül kiirásra.

.<cim>A
- A <cim> által mutatott és az azt követö byte-on elhelyezkedö vektor (alsó byte-felsö byte) értéke kerül kiirásra. Különösen decimális módban hasznos.

.<cim> - (aposztróf) A <cim> által mutatott memóriaterület elsö 8 byte-jának kiirása karakteres formában a fenti utasitások mind alkalmasak arra, hogy segitségükkel a memória tartalmát ne kiirassuk, hanem megváltoztassuk. Ez akkor következik be, ha a kód (B, A, , disassemblernél semmi) után annak megfelelö formátumu érték áll.

.T<cim1>,<cim2>,<cim3>

- A <cim1>-töl a <cim2>-1 terjedo tartományt átmásolja a <cim3>-tól kezdödö munkaterületre (transfer)

.P<cim1>,<cim2>

- segitségével egy programterületet jelölhetünk ki a következo I utasitás számára. Paraméter nelkül megadva kijelzi az aktuális programterületet.

.I<cim1>,<eltolás>

- A <cim1>-en kezdödö programrészletet - a végét .P-vel definiáltuk - a megadott értékkel eltolja. A JMP és JSR utasitásokat átcimzi, a küldö területet NOP utasitásokkal tölti fel, a .P által kijelölt programterületet kiterjeszti ill. leszükiti negativ eltolás esetén (azaz a végcim az eltolt program végcime lesz, a kezdöcim az eredeti marad).

.F(<cim>),<kód>:<operandus>

- Bizonyos utasitások keresését végzi a memóriában. A keresést egy. a <cim> által mutatott táblázat szerint végzi, amely 2-2 cimet tartalmaz, egy kezdo- és egy végcimet minden programblokkra. A táblázatot két bináris 0 byte zárja le. A keresett utasitásokat utasitáskód és/vagy operandusérték szerint azonosithatjuk. A <kód> vagy egy utasitás gepikódja, vagy a következö szimbólumok valamelyike:

  S - minden olyan abszolut cimzésmodu, amely egy cim tartalmát megváltoztatja (pl. STA, INC, ASL stb.),

  SI - ugyanez, de abszolut,indexelt cimzésmódra,

  SII - ugyanez, de indexelt indirekt és indirekt indexelt cimzésmódra,

  L,LI,LII - mint fent, de olyan utasitásokra, amelyek egy memóriabyte-ot regiszterbe toltenek vagy muveletet hajtunk végre velük (pl. LDA,EOR,ADC stb.)., tehát egy regiszter tartalmát megváltoztatják.

Ha a :<operandus>-t is megadjuk, a keresés csak az ilyen operandusu utasitásokat listázza ki disassembler formában. Az utasitás igen alkalmas programhibák felkutatására.

A következökben ismertetésre kerülö utasitások **a gépikódu programok nyomkövetését** segitik elö. Segitségükkel végrehajtás közben vizsgálhatjuk a programok hatását. Igen hasznos eszközt jelentenek még annak fenntartásásval is, hogy mivel a programfutás idöviszonyait alapvetöen megváltoztatják, bizonyos esetekben használhatatlanok ill. hamis eredményt

A **lépésenkénti végrehajtáshoz** az SM-KIT-nek szüksége van egy olyan területre, amelyet a 6510-es mikroprocesszor esetében kitüntetett fontosságu 0-ás és 1-es lap helyett használ.
Ezt tetszés szerinti memóriaterületen kijelölhetjük a

.><cim>   utasitással. A munkaterület hossza 768 byte.

A fenti utasitásra a monitor egy

```
.> AAAXXXYYYSSS--++----
..   a  x  y  s nv bdizc
```

alaku választ ad, ahol a fenti sor az aláirt regiszterek és jelzök értékét mutatja. Az elsö sor végére tetszöleges cimet irva a monitor végrehajtja a cimen lévö utasitást,a fentieknek megfelelöen kijelzi a regiszter-tartalmakat, a következö utasitás cimét és magát az utasitást disassembler formában.A RETURN megnyomásával a folyamat a következö utasitással megismétlödik.
Ha a sor végén, a <cim> helyén -<cim> formában egy végcimkifejezést irunk, a program a megadott cimig nyomozás nélkül fut, majd ott a szokásos regiszterkijelzéssel megszakad. Természetesen, ha a program nem érinti a megadott cimet, a vezérlés kicsuszik a kezünkböl, a programnál marad.

A már kipróbált - vagy például ROM-ból hivott - szubrutinok nyomozása feleslegesen lassitaná a nyomozást. Ezért lehetöség van arra, hogy a nyomozást egy JSR utasitásnál megszakitsuk és az RTS végrehajtása után automatikusan folytassuk. Ha a lépésenkénti végrehajtás során a következö utasitás egy JSR - szubrutinra ugrásutasitás, a következö sor elején egy "-R" szimbólum megadása a fenti eljárást váltja ki a nyomozásban, tehát az öt logikailag záró RTS-ig a lépésenkénti végrehajtás felfüggesztödik.

Az SM-KIT egy érdekes és igen hasznos tulajdonsága, hogy a lépésenkénti végrehajtás során a memóriakiirások ujra meg ujra végrehajtásra kerülnek. Ennek segitségével a nyomozás során figyelemmel kisérhetjük a kiválasztott memóriamezök változását. A nyomozás alatt a képernyö tetején tetszés szerinti memóriakiirásokat kérünk (A, B, '),majd a képernyö alsó részén elinditjuk a lépésenkénti végrehajtást.Minden utasitás után a memóriakiirások ismételten végrehajtódnak, igy a képernyön mindig az aktuális memóriatartalmat láthatjuk.

Az utolsó nyomozási eszköz software megszakitási pont elhelyezését biztositja a programban. A .B<cim> utasitással egy csapdát helyezünk el a megadott cimen, a program futása megszakad, a regisztereket és a jelzöket kilistázza a monitor, majd ujabb parancsra vár.

Az SM-KIT utolsó itt ismertetésre kerülö lehetosege nem a programok kipróbálását szolgálja. Az igazi helye az 5.1 fejezetben lenne, ahol többek között azzal is foglalkozni fogunk, hogy hol helyezzünk el olyan rövid assembly rutinokat, amelyeket BASIC nyelvü programból akarunk hivni. Erre egy - elég bonyolult, de sok szempontból praktikus - lehetöség magában a programban, egy BASIC sorban, a REM után elhelyezni. Az ilyen szubrutint szokás **REM-rutinnak** is hivni. Az ilyen rutin készitését segitik a következö SM-KIT parancsok:

.Z <sorszám> - A megadott sorszámu BASIC sor kezdöcimét kiirja a képernyöre. Ha a sor nem létezik, az elsö nagyobb sorszámu sor adatai kerülnek kiirásra sorszámostul. Ha a sorszám nagyobb bármelyik sorszámnál a BASIC-programban 64000-es sorszám és két 0 kerül kiirásra.

.TP,Z <sorszám> - A .P által elözöleg megadott programot a BASIC programba illeszti a megadott sorszámu sorba, mint REM-rutint. Igy pl. a

```
.P    828,833
.TP,Z  10
```

parancssorozat a 828-tól 833-ig terjedö cimeken lévö programot REM-rutinná formálva beilleszti a tárban lévö BASIC-program 10-es számu soraként.
A REM-rutinnak kötelezöen 0-val kell végzödnie (BASIC-elöirásból következöen)

Az igy létrehozható REM-rutinokkal kapcsolatos legfontosabb korlátozások:

- hossza legfeljebb 250 byte lehet,
- nem tartalmazhat önmagára hivatkozó JMP vagy JSR utasitást,
- felhivása kicsit körülményes, mivel a BASIC program szerkesztésével helye változhat a tárban, hacsak nem irjuk azokat a program legelejére.

Az SM-KIT-et, mint az assembly programozást segitö eszközt **összefoglalva** megállapitható,hogy minden eddig ismertetettnél szélesebb körü lehetöséget biztosit gépikódu programok bejáratására, elemzésére, egy jóminöségü debuggert nyujt. Hibájaként róható fel viszont, hogy nem ad eszközöket a programok létehozására, nem tartalmaz a monitor-debuggerrel egységben müködö assemblert, ezért csak jóminöségü assembler editor környezettel kiegészitve ajánlott - ugy viszont különösen - az assembly fejlesztöi tevékenységhez.

4.5 PTD interaktiv debugger

Az ebben az alfejezetben ismertetésre kerülö debugger (**PTD 6502/6510 DEBUGGER, PTERODACTYL SOFTWARE, CA USA**) egy sokkal magasabb rendü, ennél fogva bonyolultabb eszközt jelent, mint az eddigiek. Ezért talán nem érdektelen rászánni egy kicsit terjedelmesebb szövegrészt a megismerésére, mint az eddig ismertettetekkel tettük. Már elöljáróban megemlitjük, hogy ez a program nem kötödik kifejezetten a Commodore 64-hez - mint könyvünk zöme sem -, hanem minden 6502/6510 mikroprocesszorral szerelt személyi számitógéphez, igy pl. az Apple II sorozathoz és a nagyobb Commodore modellekhez is használható.

Talán nem árt már elöljáróban felsorolni a PTD debugger **jellemzöit**, amelyeket a továbbiakban részletesen is tárgyalunk. Természetesen rendelkezik a minden debuggerre szinte kötelezö minimális utasitás-csoporttal:

- memória-dump hexadecimális és ASCII formátumban,
- memória-módositás lehetösége,
- disassembler funkció
- gépikódu program lépésenkénti végrehajtása,
- tetszés szerinti információk nyomtatón is megjelenithetök,
- azonnali assembler.

Ezenkivül a következö, egyáltalán nem általános funkciókat is teljesiti:

- utasitások batch tipusu végrehajtása, ami által lehetöség nyilik tesztágy kialakitására,
- szimbólikus cimzés lehetösége a bejáratás során,
- beépitett, rezidens szimbólikus assembler,
- öndokumentáló, érthetö parancsnevek, amelyek a gyakorlott felhasználó által leröviditihetök,
- értelmes, angol nyelvü hibaüzenetek,
- kiterjesztett kifejezésfogalom,
- különbözö számrendszerbeli számok szabad keverése,
- a programok részenkénti nyomkövetési lehetösége,
- kód-foltozás (patching - jelentését ld. késöbb),
- képernyöfelosztás a nyomozás alatt felhasználói és nyomozási ablakra,
- a végrehajtott gépi ciklusok számlálása, amely elösegiti a kód utólagos hangolását,
- a joystick használata a nyomozás vezérléséhez.

4.5.1 A debugger betöltése

A lemezről LOAD "LOADER.OBJ",8,1 utasítással tölthetjük be a programot - illetőleg a töltőprogramot -, majd SYS 52000-rel indíthatjuk. A program ezután kéri a debugger számára szükséges 28 Kbyte-nyi terület kijelölését.
A választható opciók:

T	-	**felső memória** ($6000-$D000)
B	-	**alsó memória** ($800 -$7800)
<érték>	-	**kezdőcím** (tetszőleges $800-nál nagyobb érték)

Utóbbi esetben gondoskodni kell a képernyő-memória számára külön 2 Kbyte-nyi terület kijelöléséről. Erre a területre azért van szükség, mert a debugger minden beavatkozás előtt menti a képernyő-memória és a szín-memória területét, hogy a felhasználói programot a legkevésbé befolyásolja. T és B esetén a mentési pufferek kijelölése automatikus. A debugger futása alatt a felhasználói képernyőre az F3, a debugger képernyőre az F1 megnyomásával kapcsolhatunk át.

A debugger betöltése után a verem értékes részének kijelzésével jelentkezik a képernyőn.

A debuggerből az END paranccsal léphetünk ki. A vezérlés visszakerül a hívóprogramhoz, általában a BASIC-parancs módhoz.

4.5.2 Alapfogalmak

A debugger által elfogadott **számok** lehetnek decimálisak, hexadecimálisak ($-ral bevezetve), binárisak (%), negatív decimálisak (-). A karaktersorozatokat tetszés szerint aposztrófok vagy idézőjelek közé zárhatjuk. A kifejezések +, -, *, /, %OR, %AND és %XOR műveleti jeleket, valamint zárójeleket tartalmazhatnak a szokásos értelmezésben.

A debugger minden parancs végrehajtása után a sor elején vessző kiírásával (prompttal) jelzi, hogy kész a következő utasítás fogadására.

A ? **utasítást** használhatjuk - a BASIC-hez hasonlóan - kifejezések értékének kinyomtatására. A képernyőn megjelenik a kifejezés értékének decimális és hexadecimális formája egyaránt. Néhány billentyű speciális jelentéssel bír:

- RETURN — egymagában megismétli az előző parancsot,
- F1 — átkapcsol a debugger-képernyőre,
- F3 — átkapcsol a felhasználói képernyőre,
- RESTORE — "melegindítja" (ld. később) a debuggert,
- RUN — hatása megegyezik a RUN paranccsal (ld. később)

4.5.3 Program-módu végrehajtás

A FTD debugger parancsait a BASIC-hez hasonlóan kétféle módban lehet végrehajtani: parancs vagy **azonnali módban** (immediate mode) és **program-módban** (deferred mode). Elöbbi esetben az utasitás a rendszer promptja után rögtön begépelendö és azt a debugger azonnal végre is hajtja. A program-módban viszont a debugger utasitásait a BASIC programok mintájára sorszámmal bevezetve gépeljük be, amelyeket a rendszer ilyenkor csak tudomásulvesz. A végrehajtás a RUN parancs hatására történik meg. A debugger 1 Kbyte-ot tart fenn alaphelyzetben a program-módu utasitások tárolására. Ez rendszerint elegendö, de szükség esetén ki is terjeszthetö.

A program-módu utasitások kezeléséhez négy parancs áll rendelkezésre. A LIST, RUN, NEW parancsok kezelése és hatása megegyezik a BASIC megfelelöjével. A sorok törlését sorszámuk egymagában való megadásával, javitását az EDIT paranccsal végezhetjük.

4.5.4 Szimbólumok

A szimbólumok lehetnek :

- változók,
- regiszterek és jelzök nevei.

A **változók** legfeljebb 15 karakteresek lehetnek, numerikus vagy karakteres tipusuak. A numerikus változók kétbyte-os egész számokat tartalmazhatnak tetszöleges számrendszerbeli értékadással. A karakteres változók neve kötelezöen $ jellel végzödik. Csak program-módban kaphatnak értéket, amely legfeljebb 255 byte hosszu lehet.

A **fenntartott nevek** nem adhatók meg változónévként, ezek az assembly utasitások, parancsok, valamint a regiszterjelzök nevei.
Utóbbiak:

- A - akkumulátor
- X - indexregiszter
- Y - indexregiszter
- PC - utasitásszámláló
- P - állapotregiszter
- S - verem-mutató
- %C - átvitel-jelzö
- %D - decimális mód jelzö
- %I - megszakitás-érvényesités jelzö
- %N - negativ érték jelzö
- %V - tulcsordulás jelzö
- %Z - nulla jelzö

A szimbólumok értékadó utasitásokkal nyerhetnek értéket, amely a BASIC-ben megszokott formáju. Vonatkozik ez a regiszterekre és a jelzökre is! Igy az

```
X=A
```

helyes utasitás (hatása ekvivalens TAX gépi utasitáséval). A jelzök értéke magas lesz nem nulla kifejezéssel történö értékadással, ellenkezö esetben alacsony lesz. A kifejezésekben a regiszterek és jelzök szabadon használhatók. Azok értéke természetesen mindig a felhasználói program által "látott" és nem a debugger által pillanatnyilag használt valódi regiszterekkel és jelzökkel egyezik meg.

4.5.5 DOS kiegészités

A VC 1541 lemezegység vezérlésére a debugger kiegészitö parancsokat nyujt.

A DOS "$" parancs a lemezkatalógust listázza a képernyön. A DOS parancs után idézöjelek között tetszöleges, a diskvezérlönek szóló parancsot kiadhatunk, pl. DOS "S:T*" törli a T betüvel kezdödö nevü file-okat.

Programok töltésére a

```
LOAD "<filenév>""<kezdöcim>"
```

alaku utasitást használhatjuk. A parancs LD-re rövidithetö, a <kezdöcim> elhagyható, ilyenkor a lemezen tárolt kezdöcimre töltödik be a program.

Programok mentésére a

```
SAVE "<filenév>"<kezdöcim><végcim+1>
```

alaku parancs szolgál. A SAVE SV-nek is rövidithetö, a cimek megadása kötelezö.

Debugger-utasitásokat tartalmazó **program-file-ok töltése** az eltérö

```
LOADP "<filenév>"
```

utasitással (LOADP helyett LDP elég), **mentése** a

```
SAVEP "<filenév>"
```

alaku (SAVEP helyett SVP irható) utasitással történik. LOAD és LOADP utasitások végrehajtása után a debugger kiirja a program által elfoglalt utolsó byte cimét.

4.5.6 Memóriakezelés

A DUMP <kezdöcim><hossz> alaku parancs szolgál a memória egy részének hexadecimális és ASCII formáju kiirására. A <hossz> elhagyható, ekkor egy sor - 8 byte - kerül kiirásra.

Emellett használható a BASIC PEEK függvényhez és POKE utasitáshoz igen hasonló két lehetöség a memória vizsgálatára és átirására. Felhasználásuk nyilván program-módban elönyös. A szintaxis megegyezik a BASIC megfelelökkel, de szabadabb, a zárójel ill. a vesszö az utasitásban szóközzel helyettesithetö.

A <cim> alaku utasitással egy kétbyte-os értéket irathatunk ki. Az utasitás természetesen figyelembe veszi a 6510 szokásos alacsony helyiértékü byte - magas helyiértékü byte sorrendjét.

A memóriakezelés másik oldala a disassembler és assembler funkciók, tehát programok listázása a memóriából, illetve programrészletek (vagy teljes programok) beillesztése.

A **disassemblert** a

DISASM <kezdöcim><sorok száma><cim>

alaku utasitással aktivizálhatjuk. Valamennyi paraméter elhagyható. A paraméterek és alapértékük (zárójelben):

<kezdöcim>	-	a visszaforditandó program kezdöcime (az utasitásszámláló állása)
<sorok száma>	-	a listázandó utasitások száma (15)
<cim>	-	a disassembler ide helyezi (ha megadtuk) a visszaforditott programot ASCII-assembly forráskód formájában. Innen késöbb SAVE paranccsal kimenthetö lemezre. A terület végcimét DUMP utasitással kereshetjük meg.

A PTD debugger, mint már emlitettük, egy jóminöségü, kétmenetes szimbólikus assemblert tartalmaz, amely azonban csak aránylag rövid, mintegy 100 soros programok forditására alkalmas. Fö hivatása nem is az önálló programozás, hanem a kód-foltozás (ld. késöbb) támogatása. Az assembler által elfogadott nyelv szintaxisa megegyezik a szokványos formátummal. Egyetlen korlátozás, hogy program-módban csak megjegyzést tartalmazó sorok nem megengedettek.

Az **assembler** tulajdonképpen egy külön alrendszert képez a debuggeren belül. Ennek legfontosabb következménye az, hogy az assembler aktivizálása után debugger parancsok nem adhatók ki, egészen a debuggerhez való visszatérést kiváltó RET assembler direktiváig.

Az assemblert kétféleképpen használhatjuk parancs és program-módban.
Parancs-módban az

ASM <kezdöcim>

alaku parancs után a debugger megszokott vesszö (",") promptját felkiáltójel ("!") váltja fel, ez utal arra, hogy itt assembly utasitásokat vár el, a program az addigi debugger parancsok helyett. Az assembly sorokat sorszám nélkül (!) adhatjuk be. A sorszámmal beirt sorok nem kerülnek forditásra, hanem a programterületen tárolódnak. A szokásos assembly utasitások megengedettek, de az operandus mezöben a debugger alatt definiált változók használhatók, értékük az assemblerben is elérhetö. Az ASM parancsban megadott kezdöcim határozza meg azt, hogy hova kerül a leforditott program a memóriában. Ha ez elmarad, a programszámláló aktuális értékétöl kezdödik a kód lerakása. A forditás befejezése után a programszámláló az elsö szabad helyre mutat. Az assembly utasitásokon kivül két direktivát tartalmazhat a program, a DUP és a RET direktivákat. Használatukra a kód-foltozás tárgyalásánál térünk ki, de már most megemlitjük, hogy a paraméter nélkül irt RET egyszerüen lezárja az assembly forditást és visszaadja a vezérlést a debuggernek. Lássunk egy rövid példaprogramot az assembler parancs-módu használatára:

```
,ASM $C000
!SZAMX NOP
!SZY   TYA
!      RTS
!BELEP STX SZAMX
!      CPY SZAMX
!      BMI SZY
!      TXA
!      RTS
!RET
```

A programhoz nem árt egy kis magyarázat.
Az elsö sor felhivja az assemblert és közli a program kezdöcimét ($C000). A program feladata, hogy X és Y regiszterben lévö két szám közül eldöntse, melyik a kisebb és azt az A regiszterben visszaadja. A kicsit kitekert irásmódot az kivánta meg, hogy a PTD assemblerében parancs-módban nincs lehetöség elöre hivatkozásra, minden szimbólumnak a felhasználáskor már definiáltnak kell lennie. Ezért a SZAMX memóraiváltozót és az SZY programcimkét elöre kellett megadni. A szubrutin végrehajtása a BELEP ponton kezdödik. A záró RET visszaadja a vezérlést a debuggernek.

Sokkal hatékonyabban használható az assembler **program-módban**. Ekkor az ASM utasitást és az assembly programot egyaránt, mint egy PTD debugger program utasitásait kell megadni. Ebben az esetben természetesen elöre hivatkozások is használhatók a programban. Az assembler használata programmódban nagymértékben emlékeztet a PROFIMAT assembleréhez és

annak megfelelöen itt is felhasználhatjuk a parancs-nyelv feltételes vezérlésátadását makrószerü lehetöségek igénybevételére. A megoldás szinte tökéletesen megegyezik az ott bemutattal:

```
 10  Z1=$14
 20  MUTATO=2
 30  SORSZAM=1
 40  ASM $C000
 50  LDA #C1
 60  STA Z1+1
 70  LDA #0
 80  STA Z1
 90  RET
100  CIKLUS
110  ASM
120  LDY #MUTATO
130  LDA #SORSZAM
140  STA (Z1),Y
150  RET
160  IF SORSZAM<3 THEN SORSZAM=SORSZAM+1:MUTATO=MUTATO+5:
165  GOTO CIKLUS
```

Bár a program két olyan lehetöséget is tartalmaz, amelyröl még nem volt szó (cimkét és if utasitást) jelentése igy is elég világos. A ciklusszervezéssel elértük azt, hogy a középsö részt (120-140 sorszám-tartomány) csak egyszer kellett leirni. A programrészlet egyébként egy, a $C100-on elhelyezett táblázatot tölt ki. Az egymásutáni 5 byte-os bejegyzések 3. byte-jába a bejegyzés sorszámát (1-3) teszi. A programot hagyományos módon igy irtuk volna:

```
 10  Z1=$14
 20  ASM $C000
 30  LDA #$C1
 40  STA Z1+1
 50  LDA #0
 60  STA Z1+1
 70  LDY #2
 80  LDA #1
 90  STA (Z1),Y
100  LDY #7
110  LDA #2
120  STA (Z1),Y
130  LDY #12
140  LDA #3
150  STA (Z1),Y
160  RET
```

A két program által generált gépikód tökéletesen megegyezik. Visszatérve az elözö megoldáshoz, vegyük észre, hogy az ASM utasitást a második (ciklusba ágyazott) esetben paraméter nélkül hivtuk,ennek eredményeként a forditást oda folytatta, ahol az utolsó RET után félbeszakadt.Igy a generált kód folyamatos lesz.

Az assembler a következö **direktivákat** ismeri:

.BA - program kezdöcime (* helyett)
.DS - byte-ok foglalása (darabszám követi)
.BY - egy byte foglalása (kezdöértékmegadás követi)

Még ehhez az alfejezethez sorolható, bár általános érvényü a következö parancs:

PORT <egységszám>

Hatása hasonló a BASIC OPEN és CMD utasitások együttes hatásához. PORT 4 kiadása után a felhasználói program outputjai csak a printeren, a debugger outputjai a printeren és a képernyön jelennek meg. A képernyös kijelzési módhoz való visszatérést a PORT 0 utasitás váltja ki.

4.5.7 Program-nyomkövetés

Elérkeztünk a debugger azon szolgáltatásaihoz,amelyek a legfontosabbak felhasználhatósága szempontjából. A PTD debugger széles körü lehetöséget biztosit a programok nyomkövetésére, a fö problémát leginkább az eszközök közötti válogatás és a parancsok néha bonyolult szintaxisának megtanulása jelenti.

A STEP utasitás segitségével a felhasználói programot **lépésenként** hajthatjuk végre. Az utasitás formája:

STEP<lépésszám>PC<cim> DISP/NODISP/JOY

A <lépésszám> a végrehajtandó gépikódu utasitások számát jelenti. Ha nem adjuk meg, a debugger egyetlen utasitást hajt végre, ha 0-át adunk meg, 65535-öt. Ha a DISP paramétert megadjuk, minden lépés, azaz minden végrehajtott utasitás után információkat ir ki a debugger, ha NODISP-et vagy semmit sem adunk meg, csak az utolsó utasitás után - tehát ha elértük a <lépésszám> által megadott limitet - iródnak ki információk. Ez utóbbi esetben a nyomozott program futása lényegesen gyorsabb, mintegy 1000 utasitás másodpercenként.

A PC <cim> paraméter a nyomozás kezdöcimét állitja be az utasitásszámlálóba, tehát ugyanazt az eredményt nyujtja, mintha a STEP parancs kiadása elött a

PC = <cim>

értékadást elvégeztük volna. Elhagyása esetén a nyomozás kezdöértéke az utasitásszámláló aktuális értéke.

A JOY paraméter ugyanugy kiváltja minden utasitás után a kiiratást, mint a DISP, de a 2-es számu joystick-csatlakozóba dugott **joystick**-kel lehetöségünk van a végrehajtás sebességét csökkenteni ill. a gomb megnyomásával teljesen felfüggeszteni. Ezáltal lehetöség nyilik a képernyön futó információk alaposabb elemzésére.

Egy-egy végrehajtott utasitás után a DISP vagy a JOY érvényessége estén a következö sor jelenik meg:

<cim><utasitás>A<érték1>Y<érték>X<érték>P<érték><op><érték2>

ahol

<cim> - a végrehajtott utasitás cime
<utasitás> - disassemblált utasitás
<érték1> - az A regiszter értéke hexadecimálisan és ASCII-ben
<érték> - a megfelelö regiszter értéke hexadecimálisan
<op> - a kiszámitott memóriacim
<érték2> - az ezen a cimen lévö érték hexadecimálisan az utasitás végrehajtása elött.

Az aláhuzott betük a képernyön reverse módban jelennek meg.

Ha a STEP végrehajtása közben bármely billentyüt megnyomjuk, a STEP végrehajtása megszakad. Ekkor egy elöre deklarált változó, BRKFLG fogja tartalmazni a billentyünek megfelelö karakter ASCII kódját. Ezt az értéket a nyomozást vezérlö programban feltételes vezérlésátadó utasitásban használhatjuk fel.

A DISP/NODISP/JOY paraméterek addig maradnak érvényben, amig egy másikat ki nem adunk, tehát nem kell megismételni azokat a következö STEP utasitásban. Igy pl. a következö utasitássorozat:

```
,STEP PC $C000 DISP
,STEP
,
,
,
```

jelentése a következö (a magában álló promptok azt jelzik, hogy ott csak a RETURN-t nyomtuk meg, ami az utoljára végrehajtott utasitást ismétli):

- egy utasitás végrehajtása a $C000 cimröl és az információk kiirása
- a következö utasitás végrehajtása kijelzéssel
- utóbbi ismételgetése.

Megjegyezzük, hogy ez a debugger leggyakoribb és legegyszerübb használati módja.

A SUB parancs szintaxisa és a paraméterek jelentése teljesen megegyezik a STEP-ével. A különbség mindössze csak az, hogy DISP és JOY módban, ha JSR (szubrutinhivás) utasitáshoz ér, a debugger megkérdezi, vajon a felhivott szubrutint gyorsan (nyomozás nélkül) akarjuk-e végrehajtani. Ha RETURN-nel válaszolunk, a szubrutin teljes sebességgel fut le. Igy kikerülhetjük már biztonságosan letesztelt ill. operációs rendszer-beli szubrutinok felesleges lépésenkénti végrehajtását. NODISP esetén a végrehajtás módja semmiben sem különbözik STEP-étöl (bár ezt a kijelentést majd a tartományok megismerése után pontositjuk).

A teljes sebességgel végrehajtott szubrutinból visszatérve a végrehajtás az eredeti nyomozási feltételek szerint folyik tovább.

A következö, igen fontos eszköz a nyomozás során a felhasználói megszakitások alkalmazása. A megszakitást a programban egy BRK utasitás végrehajtása idézi elö, amelyet a BRK debugger paranccsal (és nem assembly utasitással!) célszerü elhelyezni. Ennek formátuma:

BRK <név><cim>

ahol a <cim> egy még nem használt azonositó, amellyel a megszakitási pontra szimbólikusan hivatkozhatunk, a <cim> azon utasitás elsö byte-jának cime, ahol a megszakitási pontot el kell helyezni. A megadott cimen lévö byte-ot a debugger egy táblázatba menti és helyére egy BRK ($00) utasitást tesz. Ha a debugger a felhasználói program teljes sebességü végrehajtása során egy BRK utasitásba ütközik, a következöket végzi el:

- visszamenti a regisztereket (amelyet a BRK hatására a mikroprocesszor elmentett a verembe),
- végrehajtja azt az utasitást, amelynek a helyén a BRK van,mintha arra egy STEP DISP utasitást adtunk volna ki,
- nem menti vissza a táblázatba kimentett utasitáskódot a BRK helyett,
- a BRKFLG változót 2-re állitja,
- program-módban áttér a következö utasitás végrehajtására,
- parancs-módban promptot ir ki és utasitásra vár.

Program-módban való végrehajtás esetén a következö utasitás elágazást hozhat létre a BRKFLG alapján, amely jelzi a megszakitás okát. A megszakitási pontot a

BRKCLR ALL
BRKCLR MEGSZ1

tipusu utasitások valamelyikével lehet törölni.Az elsö tipus valamennyi megszakitási pontot törli és a kódot helyrehozza (a kimentett utasitáskódot visszamenti). A második csak a megadott nevüre hatásos. Itt vesszövel elválasztva több megszakitási pont is felsorolható.

Az eddig elmondottak az ugynevezett software-megszakitást jelentették. A debugger ezenkivül lehetöséget ad az un. lassu megszakitási pont elhelyezésére is. Ezt használhatjuk a ROM-ban lévö programok nyomkövetésére, mivel itt a program módositásának lehetösége nem áll fenn. A megszakitási pontot a

    SLOWBRK <név><cim>

paranccsal állithatjuk be, ahol a paraméterek jelentése megegyezik a BRK parancséval. Ettöl az utasitástól kezdve a felhasználói program minden utasitása után megszakad és a debugger ellenörzi, hogy elértük-e valamelyik lassu megszakitási pontot. Ennek megfelelöen a program végrehajtása nagymértékben lelassul. Ha a debugger egy lassu megszakitási pontra ér

- üzenetben közli, hogy megszakitási pontra talált,
- a BRKFLG változót 3-ra állitja.

A megszakitási pontokat a BRKCLR-hez igen hasonló SLOWCLR-rel törölhetjük.

A beállitott megszakitási pontokat (csak a software-tipusuakat) a

    DISPLAY BRK

paranccsal kilistázhatjuk.

Ha a STEP utasitással kapott kiiratás kifutott a képernyöröl, a

    SHOW <lépésszám>

utasitással az utoljára végrehajtott, legfeljebb 128 utasitásnak megfelelö outputot ujra kinyomtathatjuk. A SHOW csak a STEP-pel nyomozott utasitásokra vonatkozik, teljes sebességgel végrehajtott kódot utólag nem lehet nyomozni vele.
A SHOW egyszerre csak egy képernyönyi outputot ir ki, RETURN megnyomásával tovább lapozhatunk.

A lépésenkénti (STEP vagy SUB utasitással kiváltott) végrehajtás során lehetöség van arra, hogy **a memória bizonyos részeit figyelemmel kisérjük.** Erre szolgál a

    WATCH <név><cim>

parancs, ahol a <név> tetszöleges mnemonikus változó, amelynek kezdöértéke a <cim> lesz. A debugger egy külön **ablakot** nyit a képernyö tetején, mégpedig minden WATCH utasitás hatására egy sort. A sorban a <cim>-töl **számitott 8 byte** értéke jelenik meg DUMP formátumban és ugy változik, ahogy a memória tartalma, a STEP vagy SUB futása során. Legfeljebb 19 WATCH változó adható meg. A változók a késöbbiek során

egyszerü értékadással uj értéket nyerhetnek, ekkor az ablakban a terület tartalma jelenik meg:

```
30  WATCH  ABLAK1  $800
40  WATCH  ABLAK2  $880
50  STEP 30 PC $900 DISP
60  ABLAK1=$830
70  STEP 30
```

A fenti programrészlet egy, a $900-as cimen kezdödö program nyomkövetését vezérli. A 30-as és 40-es sorban kétsornyi ablakot veszünk el a debugger képernyö-területéböl annak felsö részén. A legfelsö sorban a $800-$808-ig, az alatta lévöben a $880-$888-ig terjedö memóriabyte-ok tartalmanának változását kisérhetjük figyelemmel. Az 50-es utasitás vegrehajtása - 30 utasitás lépésenkénti nyomkövetése - közben a képernyö alsó 22 sorában a nyomkövetési információk futnak, a felsö két sor helybenmarad. A 60-as utasitás átállitja a legfelsö sorban látható memória-ablakot a $830-$838-ig terjedö területre. Ez lesz látható a 70-es utasitás hatására a nyomozott 30 gépikódu utasitás végrehajtása során.

A nyomozás során is, de különösen az igényeknek eleget tevö, helyesen müködö program vizsgálatánál nyujt nagy segitséget **a ciklusszámlálók használatának lehetösége.** A ciklusszámlálót a

    TIMESET <név>

vagy a

    TIMESET <név1><név2>

utasitással aktivizálhatjuk és egyidejüleg nullázhatjuk. A második alakot akkor alkalmazzuk, ha elöreláthatólag a ciklusszáma meghaladja a 65535-öt, ekkor a név2 a négybyte-os számláló felsö helyiértékü byte-jait tartalmazza. A számláló (vagy az egyidejüleg érvényben lévö számlálók) minden gépi ciklusnál eggyel, ezenkivül eggyel laphatáron történö átlépésnél és minden ugró utasitásnál. A számlálás csak a STEP vagy SUB idejére vonatkozik. A számlálók értéke a

    DISPLAY CYCLE

utasitással irhatjuk ki. Leállitásuk a

    TIMEHALT <név>

utasitással történik. A ciklusszámlálókat igen elönyösen alkalmazhatjuk a kód hangolása során, annak felderitésére, hogy a program melyik részén tartózkodik legtöbbet a vezérlés, hiszen ennek optimalizálásával jelentösen felgyorsithatjuk a program egészének lefutását. Ugyancsak jól használható ez a lehetöség a különbözö módon, ugyanarra a feladatra megirt programok futási idöszükségletének összehasonlitására (benchmark).

A nyomozáshoz tartozó utolsó parancskör a **tartományokra** vonatkozik. A tartomány a memóriaterület egy olyan része, amelyen elhelyezkedö programokat a debugger minden kérdés nélkül teljes sebességgel hajtja végre, akár STEP, akár SUB van érvényben.
A tartományokban a végrehajtás akkor is teljes sebességü, ha a STEP-et NODISP paraméterrel adtuk ki.

A tartományt egy változó kettös (!) értékadásával és a SUBRANG paranccsal definiálhatjuk:

```
<név> = <kezdöcim>
<név>.= <végcim>
SUBRANG <név>
```

A formátum önmagáért beszél. Figyeljünk fel a második - végcimet megadó - utasitásban szereplö pontra!

A tartományokat a

```
SUCLR <név>
```

vagy a

```
SUCLR ALL
```

utasitással törölhetjük. Utóbbi valamennyi tartományt törli.

A tartományokat, lassu megszakitási pontokat és ciklusszámlálókat a debugger együtt kezeli. Belölük összesen egyidejüleg 32 lehet érvényben. A már emlitett

```
DISPLAY CYCLE
```

utasitás valamennyit kiirja.

Befejezésül **egy tartomány-példa**. A

```
KERNAL = $E000
KERNAL.= $FFFF
SUBRANG KERNAL
```

utasitássorozat végrehajtása után a kernal rutinokat minden esetben teljes sebességgel hajtja végre a debugger.

4.5.8 Csak program-módban használható utasitások

Az elsö ilyen utasitás az

```
IF <feltétel> THEN <utasitások> ELSE <utasitások>
```

struktura.

A feltétel és az utasitások, valamint az utasitás-struktura jelentése a szokásos, igy felesleges részletezni. Az utasitásokat kettösponttal választjuk el az IF mindkét ágában. A feltételekben használható operátorok:

>	–	nagyobb
<	–	kisebb
>=	–	nagyobb vagy egyenlö
<=	–	kisebb vagy egyenlö
<>	–	nem egyenlö
#	–	nem egyenlö
=	–	egyenlö
AND	–	és
OR	–	vagy.

Megengedett a karakteres változók és konstansok, valamint numerikus kifejezések összehasonlitása. Ekkor a numerikus kifejezés kiértékelésre kerül és az általa hivatkozott memória-területen lévö adatok kerülnek a karakteres kifejezéssel összehasonlitásra. Vigyázat, az összehasonlitás karakteres esetben a rövidebb string hosszáig tart! Igy 'ABCD' egyenlö lesz 'ABC'-vel! Az AND és OR müveleti jelek zárójelen belül nem irhatók.

A debugger programban bárhol elhelyezhetök **cimkék.** A cimke egy utasitás helyén álló azonositó, amely csak egyszer fordulhat igy elö a programban, egyébként csak GOTO utasitás után. A cimke állhat egyedül egy sorban, vagy bárhol kettöspont után, sor közben vagy sor végén is.

A cimkére (pontosabban az öt követö utasitásra) GOTO utasitással adhatjuk a vezérlést. A GOTO IF THEN vagy ELSE ágában irva feltételtöl függö vezérlésátadást tesz lehetövé, ami különösen érdekes lehet STEP utasitás után, annak megvizsgálására, hogy a STEP hogyan zajlott le. Ez a **BRKFLG** elöre definiált változó alkalmazásával lehetséges, aminek értéke

0	–	ha a STEP utasitás rendesen befejezödött,
2	–	ha lassu megszakitási pont miatt,
3	–	ha software-megszakitási pont miatt maradt abba,
>3	–	ha bármely billentyü megnyomására szakadt meg, ilyenkor a BRKFLG a billentyünek megfelelö karakter ASCII kódját tartalmazza.

A fent elmondottakat az alábbi példa segitségével világitjuk meg:

```
10  PLUSZ = 43
20  MINUSZ = 45
30  SBETU = 83
40  LOAD "PROGRAM" $C000
50  BRK  MP  $C100
60  ? "A PROGRAMOT LEPESENKENT HAJTOM VEGRE"
```

```
 70  ? "MEGSZAKITAS  S-SEL"
 80  ? "AZ ABLAKOT   +-SZAL 8 BYTE-TAL FELJEBB"
 90  ? "             --SZAL 8 BYTE-TAL LEJJEBB"
100  ? "TOLHATOD"
110  WATCH ABLAK  $C200
120  PC = $C000
130  UJRA:STEP 0 DISP
140  IF BRKFLG=0 OR BRKFLG=3 OR BRKFLG=SBETU THEN GOTO VEGE
150  IF BRKFLG=PLUSZ THEN ABLAK=ABLAK+8:GOTO UJRA
160  IF BRKFLG=MINUSZ THEN ABLAK=ABLAK-8:GOTO UJRA
170  GOTO UJRA
180  VEGE:?"VEGE A TESZTNEK!"
```

A program lépésröl lépésre:

- 10-30 – az ASCII kódok beirása a megfelelö változókba
- 40 – a "PROGRAM" nevü gépikódu program betöltése lemezröl a $C000 cimtöl kezdödöen
- 50 – software megszakitási pont beállitása $C100-ra
- 60-100 – kezelési utasitás kiiratása
- 110 – ablak definiálása a képernyö legfelsö sorába, a $C200-$C208-ig terjedö memória-területre
- 120 – az utasitászámláló beállitása $C000-ra, a program kezdöpontjára
- 130 – a lépésenkénti végrehajtás meginditása. Az utasitásszám 65535, minden lépés után kijelzést kérünk. A nyomozást valószinüleg nem az utasitásszám-korlát, hanem a megszakitási pont fogja leállitani
- 140 – a STEP befejezödésének vizsgálata. Ha
  - az utasitásszám lejárása,
  - a $C100-on való megszakitás vagy
  - az S-betü megnyomása miatt

  következett be, a VEGE cimkére ugrunk
- 150 – ha a megszakitás oka a + megnyomása, az ablakot 8 byte-tal feljebb állitjuk és folytatjuk a nyomozást
- 160 – mint a 150, de – billentyüre és az ablak memória-területének csökkentésére
- 170 – minden egyéb megszakitást véletlen billentyümegnyomásnak tekintünk és folytatjuk a nyomozást
- 180 – program vége, bucsuüzenet.

4.5.9 Kód-foltozás

Ha a bejáratás alatt lévö programban hibát találunk, több lehetöség közül választhatunk. Az elsö és legegyszerübbnek tünö az, hogy **helyben kijavitjuk a kódot**, pl. a beépitett assembler segitségével. De ez nem olyan problémamentes, mint az elsö pillanatban látszik. Ha a javitás rövidebb, NOP utasitásokkal tölthetjük fel a felszabaduló programterületet, ha viszont hosszabb, ez az eljárás nem alkalmazható. Ilyenkor látszólag nem marad más hátra, mint **forrásszinten kijavitani a programot**, majd ujra leforditani. Ez persze elég sokáig tart. A harmadik megoldás, amiröl itt szó lesz, a **kód-foltozás.** Ekkor a javitás helyén egy ugró-utasitást helyezünk el a memória egy használaton kivüli részére, majd ott beirjuk a szükséges kiegészitést és visszaugrunk a programba. Persze eközben igen könnyü hibázni és az ilyen hevenyészett javitások által bevitt hibák felderitése általában a legkellemetlenebb feladat.

Ezért is nagyon kellemes szolgáltatása a PTD debuggernek, a kód-foltozás támogatása, ezáltal hibalehetoségeinek kiküszöbölése. A foltozás elsö lépése annak a pontnak a kijelölése, ahol a programba az uj kódot illesztjük. Ezt a

```
PATCH <név> <programcim> <uj cim>
```

utasitással tehetjük meg. A <név> a folthoz rendelt név, <programcim> ennek a pontnak a programbeli cime, <uj cim> az uj kód elhelyezésére kijelölt terület kezdöcime. Az utasitás hatására a <programcim>-en lévö byte-ot a debugger kiemeli és helyére egy BRK utasitást tesz, éppen ugy, mint a software megszakitás esetén. A kimentett utasitást a DUP utasitással vihetjük át az uj területre, ezáltal elérhetjük, hogy a kód csorbitatlan maradjon. Az uj területre a kódkiegészitést a rezidens assmblerrel fordithatjuk le.
A kódrészlet utolsó utasitása egy

```
RET <név>
```

alaku utasitás, amely a kódba egy JMP utasitást helyez el vissza az eredeti programba, a foltozási pontot követö utasitására. A kicsit bonyolult eljárást legjobban egy egyszerü példával világithatjuk meg.

Tegyük fel, hogy a következö programot kivánjuk bejáratni:

```
10 ; MEGNOVELJUK AZ AKKUMULATOR TARTALMAT
20 ; $FE-VEL ES AZ EREDMENYT KETBYTE-OSAN
30 ; AZ X-Y REGISZTEREKBE TESSZUK
40 ASM $800
50 CLC
60 ADC #$FE
70 TAY ; ALSO BYTE Y-BA
```

```
 80 LDA #0
 90 ROL A ; ATVITEL A-BA
100 TAX ; FELSO BYTE X-BE
110 RTS
120 RET
```

A programot RUN-nal leforditjuk, ezáltal a kód a $800-ra került. A szubrutin kipróbálása közben jövünk rá, hogy az azt felhivó program nem is X-Y-ban, hanem a $880-$881 cimen várja az eredményt. A javitást foltozással oldjuk meg. A program visszaforditásával:

```
DISASM $800 10
```

megtudhatjuk, hogy a TAY utasitás cime $803, a TAX-é $807. Ezek lesznek a foltozási pontok:

```
PATCH YT $803 $900     ; ELSO FOLT. PONT
PARAM=$880             ; PARAMETER
ASM $900               ; AZ ELSO FOLT FORDITASA
DUP YT                 ; ATMASOLJA A FOLT. PONTOT
STA PARAM              ; MENTES, ALSO BYTE
RET YT                 ; FOLT VEGE, FORDITAS VEGE
PATCH XT $807 $910     ; MASODIK
ASM $910               ; MASODIK FORDITAS
DUP XT                 ; MASOLAS
STA PARAM+1            ; MENTES, FELSO BYTE
RET XT                 ; FORDITAS VEGE
STEP 11 PC $800 DISP   ; TESZT
```

A foltozás egyetlen hibája, hogy a programot igy nem tudjuk kimenteni, a forrásszövegen ujra el kell végeznünk a javitást.

A debugger együtt kezeli a BRK és PATCH utasitással definiált pontokat. Megnyilvánul ez abban is, hogy azt a

```
DISPLAY BRK
```

utasitással listázhatjuk ki és a BRKCLR utasitással törölhetjük. Bánjunk óvatosan a

```
BRKCLR ALL
```

utasitással, mivel az összes foltozást is tönkreteszi!

Ha a programot teljes sebességgel kivánjuk kipróbálni, ezt az

```
EXECUTE <cim>
```

paranccsal indithatjuk el. Ilyenkor a foltozások is müködnek, mégpedig szintén teljes sebességgel. A <cim> elhagyása esetén a program az utasitásszámláló (PC) értékétöl indul. A programban elöforduló BRK utasitások segitségével ill. a RESTORE gomb megnyomásával visszakerülhetünk a debuggerbe.

4.5.10 A PTD debugger értékelése

A fentiekben egy olyan programozási eszközt ismertünk meg, amely - ha némely megoldásán vitatkozni is lehet, mégis - egy példaszerüen megoldott program-bejáratási környezetet ad a 6510-es assembly programok fejlesztésében. Kiválóan alkalmas arra, hogy a program tesztelését a programozással párhuzamosan megtervezzük, a programban vizsgálati pontokat irjunk elö, ott bizonyos feltételek teljesülését várjuk el, programozott teszt-ágyat készitsünk. Alkalmas a szubrutinok egyenkénti kipróbálására, ennek során a környezet szimulálására. Segitségével az elkészült programot elemezhetjük, hangolhatjuk, sebességi összehasonlitó vizsgálatokat végezhetünk. Nagyobb lélegzetű assembly programozási munka esetén feltétlenül javasoljuk alkalmazását.

Nem csökkenti nagymértékben a software-termék értékét néhány bántó hiányossága sem, amelyek legfontosabbjai

- a szintaxis nem következetes,
- a monitor-szolgáltatások (memóriamódositás, keresés) szegényesek,
- az INPUT utasitás rettenetesen hiányzik.

A monitor-szolgáltatások szinte teljes hiányát nyilván az okozza, hogy a debugger eredetileg APPLE mikrogépre készült, ahol a monitor ROM-ból elérhetö, de ez a Commodore-n szerény vigasz. Pótolható ez egy egyszerü monitor betöltésével, ha van erre helyünk a tárban. A többi problémára nem tudunk orvosságot.

5. Mit programozzunk, hogy programozzunk assembly nyelven?

Az eddigi fejezetek az **assembly programozás eszköztárát** igyekeztek ismertetni. Ennek a fejezetnek a célja **az assembly programozás módszertanának bemutatása.** Ez sokkal nehezebb feladat, mint az 1-4. fejezet lexikális jellegü összefoglalásai.

A fö nehézséget az okozza, hogy egyaránt kivánunk szolgálni az érdeklödö amatör és a tanuló (majdani) profi igényeinek, sok érdekes részletét fel kivánjuk villantani a programozás eme fejezetének, de meg is kivánjuk alapozni a következetes munkát.

Már az elsö fejezetben volt róla szó, de nem lehet eléggé kihangsulyozni azt, hogy maga a programozás a feladat megoldásának fontos, de nem egyetlen, söt nem is a legelsö lépése. Ha azt a - cimben feltett - kérdést kivánjuk megválaszolni, hogy mit programozzunk assembly nyelven, elöször is le kell szögezni, hogy a programozási nyelv - és az egyéb felhasználandó software-eszközök - kiválasztásának kérdése csak a probléma pontos felmérése és az igények megismerése után dönthetö el. Általánosságban kimondható, hogy csak azt programozzuk assembly nyelven, ami elkerülhetetlen. Elkerülhetetlen azért, mert a kivánt hatékonyság, speciális lehetöségek igénybevétele, stb. csak ezzel az eszközzel valósithatók meg. Ne programozzuk assembly nyelven azokat a feladatokat, vagy feladatrészeket, amelyek magasszintü nyelven is elfogadhatóan megoldhatók, mert az assembly programozás emberi és gépi idöráforditása sokszorosa a magasabb szintü feladatmegoldásénak. Ne habozzunk, ha egy feladatot részekre kell szabdalni és egyes részeit assembly, más részeit magasabb szintü nyelven (pl. BASIC) kell megoldani. A többletráforditás, amit a különbözö nyelvü részek interface-inek kidolgozására forditunk, busásan megtérül a magasszintü nyelven való programfejlesztés hatékonyságával.

A fenti mondatban elöfordult egy szó - **interface** - amelyet sok értelemben, sokféle célra használunk. Akár hardware, akár software értelemben beszélünk róla, mindig részrendszerek kapcsolatának jellemzésére szolgál. Ha kicsit képszerüen akarjuk leforditani az angol eredetit, azt irja le, hogy az egyik részrendszer milyen arcát forditja a másik felé. A programozás során lépten-nyomon programegységekkel és azok interface-ével találkozunk, amelyek pontos, hatékony megfogalmazása a feladat dekomponálását jelenti és óriási lépés a megoldás felé. Különösen fontos az interface tökéletes ismerete, ha a programot egy más által irt programrészlethez kivánjuk illeszteni, amelyre böven találunk példát ebben a fejezetben. Ebben az esetben nem az interface meghatározása - mivel abba már nincs beleszólásunk, hanem alapos megismerése a záloga a sikeres feladatmegoldásnak.

5.1 Memóriagazdálkodás

A Commodore 64 memóriafelosztásának ismerete inkább a mikroprocesszor ismertetéséhez kivánkozna, itt viszont nem hardware-leirást akarunk róla adni, inkább azt a nagyon is gyakorlati kérdést próbáljuk megválaszolni, hogy **hová helyezze el az assembly programozó programját** és adatait. Különösen fontos ez abban az esetben, ha nem önálló programot irunk, hanem - mint általában - más által irt software-hez, leggyakrabban a BASIC rendszerhez, a KERNAL-hoz és esetleg egy felhasználói BASIC programhoz is akarjuk kapcsolni programunkat. Ekkor nagyon fontos, hogy ne használjunk olyan memóriaterületet, amelyet az emlitett, együttmüködö software-ek is használnak, olyan célra, amely azok funkciójával összeférhetetlen. Ezért ebben a fejezetben memóriagazdálkodáson mindazokat az ismereteket értjük, amely a hardware, a fontosabb firmware-ek és software-ek memóriahasználatának jellemzése.

A Commodore 64 memóriája 8 Kbyte-os részekre van osztva, amelyek közül néhányan több példányban is létezhetnek. Ez azt jelenti, hogy fizikailag ugyanazon a memóriaterületen több memóriabyte (általában kettö) is található. Pontosabban, azokon a cimeken ($A000-$BFFF,$D000-$DFFF,$E000-$FFFF), ahol ROM van, RAM is van, némiképpen elrejtve. Ha az ilyen kettös memóriabyte-okat irjuk, az érték mindig az adott cimü RAM-ba Ha viszont olvassuk, akkor vagy a ROM vagy a RAM tartalmát látjuk, attól függöen, hogy az 1. cimü input-output regiszter megfelelö bitjeinek értéke milyen. A bitek és a memóriaszeletek összefüggése a következö:

0. bit	$A000-$BFFF
1. bit	$E000-$FFFF
2. bit	$D000-$DFFF

Ha a megfelelö bit értéke magas, a mikroprocesszor az adott cimen lévö ROM-byte-ot olvassa, ha alacsony, a RAM-byte-ot. A $D000-$DFFF-ig terjedö terület kezelése kicsit speciális, mivel annak egyes részein három memória-chip fedi át egymást. A ROM-on és a RAM-on kivül erre a területre esnek a négy interface-chip, a VIC II képernyökezelö, a SID hangvezérlö és a két CIA általános input/output vezérlö chip regiszterei. Ezek képesek arra -, pontosabban a VIC II képes arra -, hogy amikor szükséges, olvasni tudjon a $D000-tól a $DFFF-ig terjedö ROM-ból, amely a képernyön megjelenithetö karakterek képét tartalmazza.

A ROM-ok alatti RAM-ok használatára rövidesen visszatérünk, de elötte szóljunk egy pár szót a memória egyéb részeiröl. Nem akarjuk itt részletesen ismertetni a memória fontosabb cimeit, hiszen azt igen sok munkában megtalálhatja az olvasó (ld. a mellékelt irodalom-jegyzéket), hanem inkább a

memória kevésbé fontos részeit akarjuk feltérképezni, azokat a területeket, ahova elhelyezhetö általunk irt program.

Ha egymagában müködö gépikódu programot akarunk késziteni - ami elég ritka - **az 1. byte megfelelö állitásával rendelkezésünkre áll a teljes 64 K memória.** Ekkor is kerülendö a 0-ás és 1-s lapon a programok tárolása, hiszen - különösen az utóbbit - a processzor speciálisan használja.
Megjegyzendö, hogy az I/O chipek a $D000-$DFFF-ig terjedö memóriaterület egyes részei csak olvashatók. 64K RAM (igazi) csak a bövitö egység (cartidge) GAME és EXROM vonalainak aktivizálásával érhetö el.

Ha programunkban támaszkodunk a KERNAL rutinokra és/vagy a BASIC ROM-ra, az alattuk lévö RAM-ot már csak részlegesen és csak abban az idöpillanatban használhatjuk, amikor a fenti ROM-okra nincs szükség.

Ha BASIC programmal is együtt kivánunk müködni, a helyzet bonyolódik. Ilyenkor is szabadon használhatók:

- a nullás lapon az $FB-$FF cimek,
- ha nem használunk kazettás egységet, az $A9-$AB-ig terjedö cimek is,
- átmenetileg egy sereg byte a nullás lapon, példaként $4E-$60 és $26-$29 vagy $14-$15, amelyeket egy SYS BASIC utasitással hivott assembly rutinban elronthatunk, de a BASIC is irhatja azokat,
- $200 feletti cimen a $334-töl $3FF-ig (ha nem használunk kazettás egységet),
- a $C000-tól $CFFF-ig.

A fentieken kivül, amelyek közül a $334-$3FF rövidebb BASIC kiegészitések, a $C000-$CFFF hosszabb programok számára elönyös, ellophatunk helyet a BASIC-programok tárolására szolgáló területböl is. A BASIC -programok tárolására rendelt memóriarész ($0801-$9FFF) felosztását a következö pointerek mutatják:

$2B - $2C	-	BASIC program kezdete (rendszerint $801)
$2D - $2E	-	BASIC változók kezdete, amely általában közvetlenül a program után következik, értéke annak hosszától függ
$2F - $30	-	BASIC tömbök kezdete
$31 - $32	-	BASIC tömbök vége+1
$33 - $34	-	stringek kezdete+1 (vigyázat, ez a legmagasabb cim, amin string van, innen visszafelé terjed a $31-$32-ig), rendszerint $A000
$37 - $38	-	BASIC terület vége, rendszerint $A000

Ezen pointerek állitásával helyet szabadithatunk fel assembly programunknak. Igy pl. a $2B-$2C feljebb állitásával a $801 és a beállitott cim közötti terület teljesen szabadon használható. A BASIC memória végén a $33-$34 és a $37-$38 lejjebb állitásával nyerhetünk memóriát $A000-ig, ahol a BASIC ROM kezdödik. Egy betöltött BASIC program mögött a

$2D-$2E feljebb állitásával szabadithatunk fel memóriaterületet a régi értéktöl (a program valóságos vége) az uj értékig (a BASIC változók kezdete). Mindezen müveletek után ki kell adni egy CLR BASIC utasitást vagy egy JSR $A56E assembly utasitást, hogy a változtatásokat a BASIC interpreter tudomására hozzuk. Igy gyakori megoldás, hogy az assembly betétek a BASIC program mögött vannak, és azzal együtt töltödnek be a tárba. Ekkor a SAVE az egészet kimenti. Természetesen **a BASIC program ilyenkor már nem szerkeszthetö!**

Térjünk vissza a ROM alatt fekvö RAM használatához. Ha ezt ugy akarjuk megtenni, hogy egy BASIC programmal együttmüködjünk vagy a KERNAL-t használjuk, körülményes ide programokat tölteni. Az ilyen programrészlet futása alatt a ROM inaktiv, tehát ahhoz nem fordulhat, még hivását is egy, a normál RAM-ban lévö cimröl kell megvalósitani az 1. byte megfelelö átállitása után. Kiválóan alkalmas viszont ez a terület adattárolásra. Erdemes ebböl kizárni a $D000-$DFFF-ig terjedö részletet, amelynek hozzáférése körülményesebb, igy is 16 Kbyte tárolóterületet nyerünk, ami nem is kevés a BASIC-terület mintegy 38 K-jához viszonyitva. A következö program ezt a területet teszi elérhetövé, mégpedig BASIC programból. Ezáltal egy kicsit elöreszaladunk, belekóstolunk a BASIC-kiegészitések világába, amiröl még lesz szó.

A program feladata a BASIC terület végétöl a tár végeig terjedö RAM használata, eltekintve a $D000 és $DFFF közé esö cimektöl, a határokat is beleértve. Ezt a területet, mint egyetlen összefüggö mezöt kezeli a program, sorszámozása 1-töl addig a végcimig terjed, amelyet a PIK belépési pont 0 cimmel történö hivása ad vissza (USR(0)). A programnak három belépési pontja van. A POK nevü (SYS828) a POKE BASIC utasitáshoz hasonló, a két elvárt paraméter a memóriaterületen belüli (relativ) cim és a byte-on elhelyezkedö érték.A RESET nevü belépési pontot (SYS918) célszerü legelöször meghivni a BASIC programból, mivel ez az USR vektort a PIK belépési pontra állitja. Ezután USR(cim) alaku függvényhivással - a PEEK BASIC utasitáshoz hasonlóan - a tárolóterület adott cimü byte-jának értékét kapjuk vissza. Most lássuk a programot utasitásról utasitásra:

```
100         ;PIK-POK/ASS
110         ;FELSO TAR<BASIC TERULET VEGETOL-CFFF,E000-FFFF - >=
115         ;20K> HASZNALATA BASICBOL
120         ;POK FUNKCIO SYS 828,CIM,ERTEK-KENT HIVHATO. CIM
125         ;LEGKISEBB ERTEKE 1.
130         ;MAXIMALIS ERTEKET PIK 0 CIMMEL TORTENO HIVASA ADJA
135         ;VISSZA, ERTEK 0-255
140         ;PIK FUNKCIO USR(CIM)-KENT HIVHATO
150         ;RESET FUNKCIO SYS 918-KENT HIVHATO
160 Z1      =$14
170 FAC     =$61
180 Z2      =$A9
190 USR     =$311
200        *=$33C
```

```
210 BASVEG =$37
220 CHRGET =$73
230 GETADR =$B7F7
240 YTOFAC =$B3A2
250 FRMEVL =4AD9E
260 AYTFAC =$B391
270 POK     JSR CHRGET     ;VESSZO
280         JSR FRMEVL     ;CIM FAC-BA
290         JSR GETADR     ;FAC-BOL CIM FORMABAN Z1-BE
300         JSR NOVEL      ;Z2=Z1+BASVEG
310         JSR CHRGET     ;UJABB VESSZO
320         JSR FRMEVL     ;ERTEK FAC-BA
330         JSR GETADR     ;FAC-BOL CIM FORMABAN Z1-BE
340         LDY #0
350         LDA Z1         ;ALSO BYTE-JANAK
360         STA (Z2),Y     ;TAROLASA A MEGADOTT CIMRE
370         RTS            ;VISSZA A BASIC-HEZ
380 NOVEL   CLC ;Z1-HEZ HOZZAADJUK A BASIC VEGPOINTER ERTEKET
390         LDA BASVEG
400         ADC Z1
410         STA Z2
420         LDA BASVEG+1
430         ADC Z1+1
440         STA Z2+1
441         CMP #$D0       ;IF Z2>D000
443         BMI RETNOV
445         CLC
447         ADC #$10       ;Z2=Z2+$1000
449         STA Z2+1
450 RETNOV  RTS            ;VISSZA
460 PIK     JSR GETADR     ;FAC-BOL USR PARAMETEREKENT MEGADOTT
465                        ;CIM Z1-BE
470         LDA Z1         ;HA Z1=0, A MAX CIMET KERDEZIK
480         BNE PIK1
490         LDA Z1+1
500         BEQ MAXCIM
510 PIK1    JSR NOVEL      ;Z2=Z1+BASVEG
520         LDY #0
530         LDA #53        ;ADATOK ELOKESZITESE
540         SEI            ;MEGSZAKITAS MASZKOLASA
550         STA 1          ;ROM KIKAPCSOLASA
560         LDA (Z2),Y     ;ERTEK KIOLVASASA
570         TAY            ;Y-BA
580         LDA #55        ;ROM VISSZA
590         STA 1
600         CLI            ;MEGSZAKITAS ENGEDELYEZESE
610 RET     JMP YTOFAC     ;Y FAC-BA ES VISSZA A BASIC-HEZ
620 MAXCIM  SEC            ;$CFFF-BASVEG
630         LDA #$FF
640         SBC BASVEG
650         TAY            ;ALSO BYTE
660         LDA #$CF
670         SBC BASVEG+1
730         CLC            ;+$2000
740         ADC #$20       ;FELSO
750         JMP AYTFAC     ;FAC-BA ES VISSZA A BASIC-HEZ
760 RESET   LDA #PIK<      ;USR -> PIK
770         STA USR
780         LDA #PIK>
```

270-290. sor - A POK belépési pontnál az elsö feladat a paraméterátvétel. A CHRGET hivása átlépi a BASIC szövegben szereplö soronkövetkezö karaktert, ami a SYS828 után álló vesszö. A FRMEVL interpreter rutin, mint a későbbiekben részletesen is tárgyaljuk majd, az elsö paraméter (a cim) értékét tölti lebegöpontos formában a FAC lebegöpontos akkumulátorba, amelyet a következö GETADR hivás konvertál kétbyte-os cimmé és helyez Z1-be ($14-$15).

300. sor - A NOVEL szubrutin, amelyet alább még megvizsgálunk, ezt az értéket megnöveli a BASIC terület végmutatójával, ami a $37-$38 cimen található és értéke alaphelyzetben $A000.

310-330. sor - Itt az értéket - második paraméter - helyezzük az elöbbiekben megismert módon Z1-be.

340-360. sor - Z1 tartalmát lerakjuk a kiszámitott cimre.

370. sor - POK szubrutin vége, visszatérünk a hivó BASIC programba.

380-450. sor - A már emlitett NOVEL szubrutin Z1+BASVEG-et helyezi el Z2-ben.

460. sor - Az USR-rel hivott PIK belépési pont. Az USR hivása esetén az interpreter már elöre gondoskodik a zárójelben lévö érték FAC-ba töltéséröl, igy csak a második lépés, a GETADR-ral történö konvertálás marad.

470-500. sor - Megvizsgáljuk Z1 értéket. Ha az kétbyte-os nulla, a megállapodás szerint a legnagyobb megadható cimet kérdezik,MAXCIM-re ugrunk.

510. sor - A már ismert NOVEL hivása, az abszolut cim kiszámitása BASVEG hozzáadásával.

520-550. sor - Itt történik a ROM-ok kikapcsolása. Az akkumulátorba betöltjük az 1. byte szükséges értékét, maszkoljuk a megszakitásokat és átállitjuk az 1. cimü byte-ot. A megszakitás maszkolására azért van szükség - tulajdonképpen ez csak a KERNAL-ROM-ra vonatkozik -, mert egy érkezö megszakitás a kikapcsolt részre adná a vezérlést, ahol most nem a megszakitás-kezelö szubrutin, hanem a mi adataink vannak.

560-570. sor - Az adatbyte kiolvasása és áttöltése az Y regiszterbe.

580-600. sor - Visszakapcsoljuk a ROM-ot és engedélyezzük a megszakitást.

610. sor - Ez a szubrutin utolsó utasitása,de ez elég sok magarázatot kiván. Mivel a kiolvasott értéket a FAC-ba kell helyeznünk a visszatérés elött, itt is a megfelelö interpreter rutint hivjuk meg a feladat elvégzésére, ahelyett, hogy megirnánk ujra. Ez a $B3A2-n lévö YTOFAC-nak nevezett szubrutin, amely az Y regiszter értékét lebegöpontosra konvertálva a FAC-ba tölti.Erdemes felfigyelni egy, az assembly programozásban szokásos fogásra. A szubrutint, amelynek hivása a mi szubrutinunk utolsó utasitása, nem JSR-rel hivtuk meg, hanem JMP-pal ugrunk rá. Az interpreter-rutin végén lévö RTS igy nem a mi programunkra, hanem egyenesen a minket hivó programhoz tér vissza, hiszen annak visszatérési cime van a verem tetején. Ez az eljárás, amely nem nagyon strukturált, és hatása a programból csak némi gyakorlattal olvasható ki, igen hatékony, mivel megtakaritunk vele egy cimelmentési-visszamentési müveletet és 2 byte-ot a veremben, ami néha nem elhanyagolható szempont.

620-740. sor - A már hivatkozott MAXCIM programrész feladata a megengedett maximális cim visszaadása. Ezt a $CFFF-BASVEG+$2000 kifejezés adja meg. A $CFFF-BASVEG a $D000-ig, a $2000 az azon felül rendelkezésre álló byte-ok száma.

750. sor - A kiszámitott értéket, amelyet Y-ba (alsó helyiértékü byte) és A-ba (felsö helyiértékü byte) helyeztünk, az AYTFAC interpreter rutin az elözökben használt YTOFAC-hoz hasonlóan - de két byte-ból összerakva - helyezi el a FAC-ban, ami az USR függvény visszatérési értékét kell, hogy tartalmazza a szubrutin lefutása után.

760-800. sor - A RESET szubrutin feladata csupán az, hogy PIK cimét az USR vektorba ($311) töltse.

A következökben ugyanezt a területet egy kicsit bonyolultabban, de sok esetben praktikusabban használjuk fel. A sortároló rutin ugy müködik, mint egy FIFO-verem, azaz egymásután tölthetjük bele a byte-okat, majd a belehelyezés sorrendjében ki is szedhetjük öket anélkül, hogy a soronkövetkezö byte cimét tudnunk kéne, azt a szubrutin maga tárolja. Elöször nézzük magát a programot:

```
110        ; MC RUTIN BASIC SORTAROLOHOZ. TAROLAS BGADR-$CFFF,
115        ; $E000-$FFFF (>=20K)
120        ; PUSH FUNKCIO USR(ERTEK)-KENT HIVHATO, 0 HA OK, -1
125        ; HA NEM (TUL SOK PUSH)
130        ; PULL FUNKCIO USR(0)-KENT HIVHATO, ERTEK HA OK, -1
135        ; HA NEM (TUL SOK PULL)
140        ; SETIN FUNKCIO SYS945,BGADR-KENT HIVHATO, ADATIRANY:
145        ; PULL
150        ; SETOUT FUNKCIO SYS977-KENT HIVHATO, ADATIRANY: PUSH
160 Z1     =$14
170 FAC    =$61
180 Z2     =$A9
190 USR    =$311
200        *=$33C
205 CHRGET =$73
210 GETADR =$B7F7
220 YTOFAC =$B3A2
230 FRMEVL =$AD9E
240 ILEGAL =$B248
250 PUSH   JSR ADRINC       ; KOVETKEZO CIM
270        JSR GETADR       ; USR-BEN MEGADOTT ERTEK FAC-BOL
272                         ; Z1-BE
275        LDY #0
280        LDA Z1
290        STA (Z2),Y       ; TAROLASA A KOVETKEZO CIMRE
300        LDA #0           ; FAC=0 -> OK
302        STA FAC
304        STA FAC+1
306        JMP RETOK        ; VISSZA A BASIC-HEZ
310 PULL   LDA NXTADR       ; VEGE A BEIRT ERTEKEKNEK PRINT
320        CMP ENDADR
330        BNE ADR3
340        LDA NXTADR+1
350        CMP ENDADR+1
360        BEQ OVER1        ; IGEN -> HIBAJELZESSEL VISSZA
370 ADR3   JSR ADRINC       ; CIMET NOVEL
380        LDY #0
390        LDA #53          ; ADATOK ELOKESZITESE
400        SEI              ; MEGSZAKITAS MASZKOLASA
410        STA 1            ; ROM KIKAPCSOLASA
420        LDA (Z2),Y       ; ERTEK KIOLVASASA
430        TAY              ; Y-BA
440        LDA #55          ; ROM VISSZA
450        STA 1
460        CLI              ; MEGSZAKITAS ENGEDELYEZESE
470 RET    JMP YTOFAC       ; Y FAC-BA ES VISSZA  A BASIC-HEZ
480 ADRINC INC NXTADR       ; CIMET NOVEL
490        BNE ADR1
500        LDA NXTADR+1
510        CLC
520        ADC #1
530        BEQ OVERFL       ; CIM=0000 -> MEMORIA BETELT
540        CMP #$D0         ; HA A CIM >=$D000 ES ,$E000
550        BCC ADR2
560        CMP #$E0
```

```
570          BCS ADR2
580          CLC
590          ADC #$10         ; AKKOR HOZZAADUNK $1000-T
600 ADR2     STA NXTADR+1     ; UJ CIM FELSO BYTE
610 ADR1     LDA NXTADR       ; UJ CIMET Z2-BE
620          STA Z2
630          LDA NXTADR+1
640          STA Z2+1
650          RTS
660 OVERFL   PLA              ; VISSZATERESI CIM VEREMBOL
670          PLA
680 OVER1    LDA #$81         ; FAC=-1
690          STA FAC
700          LDA #$80
710          STA FAC+1
715          STA FAC+5
720          LDA #0
730 RETOK    STA FAC+2        ; VISSZA A BASIC-HEZ
740          STA FAC+3
750          STA FAC+4
760          RTS
770 SETOUT   LDA #PUSH<       ; USR -> PUSH
780          STA USR
790          LDA #PUSH>
800          STA USR+1
810          JSR CHRGET       ; TERMINATOR
812          JSR FRMEVL       ; CIM -> FAC
814          JSR GETADR       ; FAC -> Z1 ($14/$15)
816          LDA Z1
818          STA BGADR
820          LDA Z1+1
822          STA BGADR+1
842 SETOU1   LDA BGADR        ; NXTADR=BGADR
844          STA NXTADR
846          LDA BGADR+1
848          STA NXTADR+1
850          RTS
860 SETIN    LDA #PULL<       ; USR -> PULL
870          STA USR
880          LDA #PULL>
890          STA USR+1
900          LDA NXTADR       ; UTOLSO CIM ELMENTESE
910          STA ENDADR
920          LDA NXTADR+1
930          STA ENDADR+1
940          JMP SETOU1       ; ES NXTADR BEALLITASA AZ ELEJERE
950 NXTADR   .BYTE 0,0
960 ENDADR   .BYTE 0,0
970 BGADR    .BYTE 0,0
```

A programot érdemes részletesebben is megszemlélni. Négy belépési pontja van BASIC-ből:

1. A programot - amely a $33C-töl indul - elöször a SYS 945,KC BASIC utasitással hivhatjuk fel. A hivás a SETOUT belépési pontnak felel meg, amely elökésziti az értékek beirását a memóriába s egyuttal tudomására hozza az assembly programnak a rendelkezésére álló terület kezdöcimét (KC),ez egyszerü esetben $A000 lehet.
2. Ezután tetszés szerint sokszor hivhatjuk a programot az USR(E) függvény aktivizálásával, ahol E az elhelyezni kivánt byte értéke. A függvény értéke 0, ha a tárolás rendben ment, -1, ha átléptük a memória felsö határát ($FFFF).
3. Az értékek kiolvasása a SYS 987 BASIC utasitással készithetö elö.
4. Az egymásután hivott USR(0) függvények visszaadott értéke a soronkövetkezö byte értéke, ill. -1, ha tulléptük a beirt byte-ok mennyiségét. A visszaadás FIFO tipusu,tehát az elsönek beirt értéket kapjuk vissza az elsö olvasáskor.

Vegyük sorra a belépési pontokat:

770-800. sor - A BASIC USR függvénynek egy vektor felel meg, amelynek cime $311-$312.Ha aktivizáljuk az USR függvényt a BASIC programban, a zárójelben lévö kifejezés értéke a FAC lebegöpontos akkumulátorba kerül ($61-$65, $66 a FAC elöjelét külön is tárolja) és meghivódik a $311-$312 vektor által hivatkozott gépikódu program. A program output paraméterét,amely szintén lebegöpntos szám, a FAC-ban adja vissza. Ezek a programsorok a $311-$312-be a PUSH cimke által hivatkozott cimet töltik, igy a következö USR hivás oda adja majd a vezérlést.

810-840. sor - A SYS utasitással hivott gépikódu programrészlet futásakor a BASIC interpreter által olvasandó következö karakter (amelyet a $73 cimü CHRGET szubrutin ad meg) a gépikódu program cimét meghatározó kifejezést követö byte. Jelen esetben ez egy vesszö. Ezt a CHRGET hivásával lépjük át, ellenörzés nélkül.Ezután egy numerikus kifejezést várunk el,amelynek értéke a felhasználandó terület kezdöcime. A BASIC interpreter $AD9E cimen kezdödö szubrutinja, amely a FRMEVL (Formula Evaluation) nevet viseli, a következöket hajtja végre: meghatározza a kifejezés értékét és azt elhelyezi a FAC lebegöpontos akkumulátorban. Ez nagyon jó, de nekünk a cim kétbyte-os egész alakra konvertált alakjára van szükségünk. Ezt a konvertálást teljesen felesleges megirnunk,mivel éppen ezt

az interpreter $B7F7 kezdöcimü GETADR szubrutinja, amely a FAC értékét kétbyte-os, cim formátumu (alacsony byte - magas byte) egésszé alakitja és elhelyezi a $14-$15 cimeken (Z1). Ezt az értéket áttöltjük a saját területen lévö BGADR és NXTADR változókba. Utóbbi majd a tároló utolsó foglalt cimét mutatja.

850. sor - Visszatérés a BASIC programba.

250. sor - Ezt a belépési pontot irtuk a $311-$312 vektorba, tehát az USR hivásakor ide kerül a vezérlés. Ekkor a zárójelben lévö kifejezés értéke FAC-ban van.Elöször a tárolás pointerét (NXTADR) növeljük meg egy byte-tal. ezt az ADRINC szubrutin végzi, amely a 480-650 sorokban helyezkedik el.

480-520. sor - Itt növeljük a kétbyte-os NXTADR mutatót. A kisebb helyiértékü byte-ot helyben, a nagyobb helyiértéküt (ha szükséges, laphatáron) az akkumulátorban.

530. sor - Ha a felsö byte is nulla, kimeritettük a tárat, több értéket nem tudunk elhelyezni, az OVERFL cimkére ugrunk.

540-590. sor - Ha a cim $D000 és $DFFF közé esik, átlépjük ezt a tartományt ugy, hogy a magasabb helyiértékü byte-ot $10 hozzáadásával $E0-ra növeljük. Azoknak, akik a vizsgálatot (>= $D0 és < $E000) felesleges óvatosságnak tartják, hiszen a $D0-ra való egyenlöség-vizsgálat is elegendö lenne, ajánlom tanulmányozásra Murphy törvényeit...

600-650. sor - A magas helyiértékü byte tárolása, majd az aktuális cim áttöltése Z2-be ($A9), amely az indirekt cimzésmódhoz kell, mint nullás lapon lévö pointer. Ez a kétbyte-os mutató most a tárolásra használt byte cimét tartalmazza.Ezután visszatér a hivás helyére.

660-760. sor - Mielött visszaugranánk, nézzük meg gyorsan a tulcsordulás esetét. Elöször két értéket kiveszünk a veremböl, mivel itt ki fogunk hagyni egy szubrutinból való visszatérési (RTS) utasitást és ennek vektora még a stack-ben van.Persze ez nem szép megoldás, nem strukturált, nem illendö, csak hatékony. Ezután lebegöpontos -1-et töltünk a FAC-ba ($81,$80,0,0,0) és $80-at a FAC külön elöjelébe, amely jelzi, hogy a szám kisebb, mint 0. A rutint záró RTS a BASIC programba visz vissza.

270-280. sor - Mivel a függvény paramétere már a FAC-ban van, ezt a már jól ismert GETADR szubrutinnal visszük $14-$15-re (Z1). Ebből csak az alsó byte-tal foglalkozunk - tehát a 256-os osztás maradékát vesszük, ha a kifejezés nagyobb 256-nál -. Az Y regiszterbe nullát töltünk az indirekt indexelt cimzésmód módositó értékeként.

290. sor - Ez a lényeg, a byte-ot lerakjuk a Z2 által mutatott helyre.

300-306. sor - A visszatéréshez a 0 függvényértéket ($61-$66-ig csupa 0) tesszük el és a RETOK cimkén át - kihasználva az ott már egyszer leirt utasitásokat - visszatérünk a hivó BASIC programhoz.

860-940. sor - Ez a belépési pont (SETIN) az olvasás előkészitése. Az első lépés az USR vektorának átállitása programunk PULL cimkéjére. A NXTADR most a legutoljára beirt byte cimét mutatja, ezt elmentjük ENDADR-be, majd a SETOU1 cimkére ugrunk, ahol ismét a BGADR tartalmát - a terület kezdőcimét - töltjük NXTADR-be és visszatérünk a BASIC programhoz.

310-360. sor - Az ujonnan beállitott USR belépési pont. Az első lépés annak vizsgálata, hogy vége van-e a beirt értékeknek -ENDADR alapján-, azaz az előző lépésben visszaadott byte már az utolsó byte volt-e. Ha igen, visszatérés az OVER1 cimkén át, amely megegyezik OVERFL hatásával, kivéve a veremmutató kettővel való növelését, hiszen az itt felesleges (söt, ártalmas) lenne.

370-380. sor - Az ADRINC hivásával a már ismert rutinnal növeljük NXTADR értékét. Itt tulajdonképpen az ADRINC-ben lévő vizsgálat, hogy NXTADR meghaladja-e $FFFF-et, felesleges, hiszen az ENDADR-ral való összehasonlitás előbb jelez, de igy nem kellett egy külön szubrutint irni, meg egyébként is (Murphy...). Az Y regiszter nullára állitása itt is az indirekt indexelt cimzést szolgálja.

390-470. sor - lásd az előző programhoz fűzött megjegyzéseket.

Ezzel kis programunk ismertetésének a végére értünk, amely több szempontból is tanulságos volt. Először is szemléltette az elrejtett ROM-ok használatát. Másodszor jó példáját adta a BASIC programmal együttműködő program interface-ének, felvonultatva a két fontosabb változatot:

- SYS utasitás esetleges paraméter-átadással, és
- USR függvény alkalmazása és paraméter-kezelése.

Harmadszor bemutatta azt, hogyan kell a BASIC-interpreter rutinokat felhasználni az assembly nyelvü programbetétekben a BASIC rendszerrel való kapcsolattartásra.

Ebben a programban több olyan dologról is szó esett, amelyet a későbbiekben részletesebben fogunk tárgyalni, de talán hasznos ezeket az elemeket elöször egy gyakorlati alkalmazásban látni.

A fenti program egy speciális feladat megoldására szolgál, arra, hogy a BASIC-terület végétöl a rendelkezésre álló helyen, amely 20 Kbyte-tól akár 55 Kbyte-ig is terjedhet, hogyan lehet egy puffertárolót létesiteni. Jó programozási gyakorlat ennek átirása kicsit más - hasznos - feladatok megoldására, pl. fentiekhez hasonló rutinokká, de nem egy-byte-os értékek,hanem pl. BASIC-lebegöpontos számok (5 byte) tárolására (segitségül: az FRMEVL-lel átvett kifejezés értékét FAC-ból tölthetjük a memóriába és viszont).

5.2 Assembly = szerelés

Kicsit kitekerve az assembly szó jelentését - pontosabban az eredeti jelentését véve alapul - azt mondhatjuk, hogy az assembly programozás nem más, mint alkatrészek összeszerelése. Aki egy kicsit is belekóstolt már az assembly programozásba, biztosan észrevette, hogy általában ugyanazon vagy roppantul hasonló elemeket, néhány utasitásból álló programrészleteket kell kombinálni. Nyilván ezért is kézenfekvö - ha lehetöségünk van rá - a makróassemblerek használata.

Ismerkedjünk most meg az assembly programozás néhány egyszerü épitökövével, alkatrészével! Hogy a programrészletek jelentése anélkül is nyilvánvaló legyen, hogy sokat magyaráznánk, baloldalt egy BASIC programrészletet, jobboldalt a megfelelö vagy nagyon hasonló assembly utasitássorozatot adjuk meg.

Ertékadások:

```
AA=12                         LDA #12
                              STA AA

B=C                           LDA B
                              STA C
```

Aritmetika egybyte-os értékekkel:

```
SM=SM+20                      LDA SM
                              CLC
                              ADC #20
                              STA SM

SM=SM-13                      LDA SM
                              SEC
                              SBC #13
                              STA SM

Z=Z+F-2                       LDA Z
                              CLC
                              ADC F
                              SEC
                              SBC #2
                              STA Z

Z=Z*2                         ASL Z
```

```
Z=Z*10                                LDA Z
                                      ASL A    ; *2
                                      ASL A    ; *2 -> *4
                                      ASL A    ; *2 -> *8
                                      CLC
                                      ADC Z
                                      CLC
                                      ADC Z    ; 8*Z+Z+Z
                                      STA Z

N=N+1                                 INC N

N=N-2                                 DEC N
                                      DEC N
```

Programstrukturák:

```
FOR N=1 TO 5                          LDX #1
...                             CFEJ  ...
NEXT N                                ...
                                      INX
                                      CPX #5
                                      BNE CFEJ

FOR N=0 TO 100 STEP 4                 LDX #0
...                             CFEJ  ...
NEXT                                  ...
                                      CLC
                                      TXA
                                      ADC #4
                                      TAX
                                      CPX #100
                                      BNE CFEJ

FOR N=S+8 TO K                        LDA S
...                                   CLC
NEXT                                  ADC #8
                                      TAX
                                CFEJ  ...
                                      INX
                                      CPX K
                                      BNE CFEJ
```

Megjegyzés: a ciklus legalább egyszer lefut!

```
FOR N=10 TO 1 STEP-1                  LDX #10
...                             CFEJ  ...
NEXT                                  ...
                                      DEX
                                      BNE CFEJ

IF K<>0 THEN GOTO 500                 LDA K
                                      BNE CIM500
```

```
IF K+3 < F THEN GOTO 500          LDA K
                                  CLC
                                  ADC #3
                                  CMP F
                                  BMI CIM500

IF FF >= 0 THEN GOTO 500          LDA FF
                                  BPL CIM500

IF K > 0 THEN K=13                LDA K
                                  BMI NEXT
                                  BEQ NEXT
                                  LDA #13
                                  STA K
                           NEXT   ...

IF N=0 THEN S=-S                  LDA N
                                  BNE NEXT
                                  LDA S
                                  EOR #$FF ; LOGIKAI KOMPLEMENS
                                  CLC
                                  ADC #1
                                  STA S    ; KETTES KOMPLEMENS
                           NEXT   ...
```

Megjegyzés: S egybyte-os kettes komplemens ábrázolású szám.

```
IF K < 20 AND K > 0               LDA K
   THEN GOTO 100                  CMP #20
                                  BPL NEXT
                                  CMP #0
                                  BPL CIM100
                           NEXT   ...

IF K < F OR K = 32                LDA K
   THEN GOTO 100                  CMP F
                                  BMI CIM100
                                  CMP #32
                                  BEQ CIM100
```

Kétbyte-os aritmetika:

```
N=N+1                             INC N
                                  BNE FOLYT
                                  INC N+1
                          FOLYT   ...
```

Megjegyzés: a felső byte növelése akkor szükséges, ha az alsó byte 0-ra ugrik.

```
N=N-1                             BIT N
                                  BNE FOLTT
                                  DEC N
                          FOLYT   DEC N+1
```

Az alsó byte-ot nem DEC, hanem SBC utasitással csökkentettük, mivel a DEC nem állitja a C jelzöt, amelyet a 0-ról csökkenés figyelésére használhatunk. A Z jelzö nem jó, mert nem abban a lépésben kell a felsö byte-ot csökkenteni, amikor az alsó byte 0, hanem a következöben. A programrészletet még sokféleképpen meg lehet irni, ez is egy jó megoldás.

NN=C+B (C és B egybyte-osak, NN kétbyte-os)

```
        LDA #0
        STA NN+1
        CLC
        LDA C
        ADC B
        STA NN
        BCC FOLYT
        INC NN+1
FOLYT   ...
```

Megjegyzés: az alsó byte-ba az egybyte-os összeg, a felsöbe a C jelzö értéke kerül.

NN=KK+B (NN és KK kétbyte-osak, B egybyte-os)

```
        LDA KK+1
        STA NN+1
        CLC
        LDA KK
        ADC B
        STA NN
        BCC FOLYT
        INC NN+1
FOLYT   ...
```

NN=KK+LL (mind kétbyte-os)

```
        CLC
        LDA KK
        ADC LL
        STA NN
        LDA KK+1
        ADC LL+1
        STA NN+1
```

Megjegyzés: ez a legegyszerübb, hiszen a C jelzö és az ADC utasitás éppen a kétbyte-os összaadást segiti.

NN=KK-B (NN és KK kétbyte-os, B egybyte-os)

```
        LDA KK+1
        STA NN+1
        SEC
        LDA KK
        SBC B
        STA NN
        BCS FOLYT
        DEC NN+1
FOLYT   ...
```

```
NN=KK-LL (mind kétbyte-os)                    SEC
                                              LDA KK
                                              SBC LL
                                              STA NN
                                              LDA KK+1
                                              SBC LL+1
                                              STA NN+1

NN=NN*2                                       ASL NN
                                              ROL NN+1
```

Megjegyzés: ASL egy bittel balra lépteti a számot, az utolsó bit helyére 0 lép be, ez kettövel való szorzást jelent. A ROL megegyezik ASL-lel, de az elözö lépésben kilépett átvitelt (C jelzöböl) beviszi a felsö byte legalsó helyiértékére. Néggyel való szorzás a fenti két utasitás meg s.i.t. Nem kettö hatványával való szorzás léptetések és hozzáakások kombinálásával lehetséges, nagyobb vagy változóban tárolt számmal való szorzásnál ciklust érdemes szervezni.

Programstrukturák kétbyte-os számokkal:

```
IF NN <> 0 THEN GOTO 100                      LDA NN
                                              ORA NN+1
                                              BNE CIM100

IF NN <> KK THEN GOTO 100                     LDA NN
                                              CMP KK
                                              BNE CIM100
                                              LDA NN+1
                                              CMP KK+1
                                              BNE CIM100

FOR NN=0 TO 4200 (4200=$1068)                 LDA #$69
...                                           STA VEG
NEXT                                          LDA #$10
                                              STA VEG+1
                                              LDA #0
                                              STA NN
                                              STA NN+1
                                        CFEJ  ...
                                              ...
                                              INC NN
                                              BNE F1
                                              INC NN+1
                                        F1    LDA NN
                                              CMP VEG
                                              BNE CFEJ
                                              LDA NN+1
                                              CMP VEG+1
                                              BNE CFEJ
```

Ugy érezzük, ennyi épitöelem ismertetése már elegendö ahhoz, hogy az olvasó felismerje a bonyolultabb algoritmusok képzését ezekböl. Erdemes programozni az ilyen egyszerü utasitássorozatok felhasználásával még akkor is, ha az ezekböl összeállitott program nem a leghatékonyabb, hiszen **a müködö programot utólag könnyebb hangolni, mint egy bonyolult**, elemeiben is **uj megoldást letesztelni.**

A fentiek illusztrálására lássunk egy aránylag hosszu, de igen egyszerü programot, amely az assembly programozásban néhány gyakran elöforduló épitökövet használ és egyes részfeladataival szintén sürün találkozunk. **A program memóriadumpot ir ki a képernyöre** a monitorok megszokott stilusában, egy sorba 8 byte hexadecimális és ASCII értékét is kiirva. A sor elején a sorban kinyomtatott elsö byte cimét irja ki hexadecimálisan, amelyet egy kettöspont követ. Bár a feladat egyszerü, mégis aránylag sok, önmagában kerek részfeladatból áll, igy pl.:

- hexadecimális érték beolvasása és konvertálása kétbyte-os binárissá,
- ciklus szervezése alsó határtól felsö határig 8 lépésközzel,
- bináris egy- és kétbyte-os érték konvertálása hexadecimálissá,
- kiirandó karakterekböl a speciális karakterek kiszürése,
- billentyüzetröl olvasás,
- képernyöre irás.

Utobbi két részfeladattal egy kicsit elöreszaladunk a következö fejezet témakörébe, de megértésük most sem okoz nehézséget. Lássuk elöször a program teljes szövegét:

```
100         ;DUMP PROGRAM -KEZDOCIME 49170=$C012
110 CR      =$00
120 SPACE   =$20
130 KERDO   =$3F
140 CHRIN   =$FFCF
150 CHROUT  =$FFD2
160        *=$C000
170 N       =$59
180 NK      =$5B
190 MUT     .BYTE 0
200 ALAP    .BYTE 0
210 HX      .TEXT "0123456789ABCDEF"
220         ;* * * * * * * * * * * * * * * * * * * * * * * * * *
230 KEZDET JSR ERTEK    ;KEZDOCIM BEOLVASASA
240         BNE HIBA    ;HIBA VOLT
250         LDA N       ;ELSO CIM
260         STA NK      ;EZ AZ ALSO HATAR -> NK-BA
270         LDA N+1
280         STA NK+1
290         TXA         ;VEGJEL
300         CMP #CR
310         BEQ KIIR    ;HA AZ ELSO ERTEK CR-REL VEGZODO TT, NINCS MASIK
320         JSR ERTEK   ;FELSO HATAR BEOLVASASA
330         BNE HIBA    ;HIBA VOLT
340 KIIR    JSR DUMP
350 VEGE    RTS         ;VISSZA A BASIC-HEZ
360         ;* * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
370 ERTEK  LDA #0        ;ALAP=0 JELZI, HOGY AZ ELSO SZAMJEGY KOVETKEZIK
380        STA ALAP
390 SPC    JSR CHRIN     ;LEOLVASSUK AZ ELEJEROL A BEVEZETO SPACE-EKET
400        CMP #CR       ;HA ROGTON CR JOTT. HIBA
410        BEQ SYNTAX
420        CMP #SPACE    ;HA SPACE, LASSUK A KOVETKEZOT
430        BEQ SPC
440        BNE VIZSG     ;HA NEM, EZ MAR ERTEKES JEGY
450 CIKLUS JSR CHRIN     ;ERTEK KOVETKEZO KARAKTERE
460        CMP #CR       ;HA CR, VEGE AZ ERTEKNEK
470        BEQ ENDE
480        CMP #SPACE    ;HA SPACE, AKKOR IS
490        BEQ ENDE
500 VIZSG  CMP #'0       ;HA  < "0" HIBAS
510        BMI SYNTAX
520        CMP #'9       ;HA >="0" ES <="9" AKKOR SZAMJEGY
530        BMI SZAM
540        BEQ SZAM
550        CMP #'A       ;HA < "A" AKKOR HIBA
560        BMI SYNTAX
570        CMP #'G       ;HA >= "G" AKKOR HIBA
580        BPL SYNTAX
590 BETU   SEC           ;BETU JOTT
600        SBC #'A-10    ;KIVONJUK "A" KODJAT< HO[illegible]DUNK 10-ET
610        JMP OSSZEG    ;ES HOZZAADJUK AZ EDDIGIHEZ
620 SZAM   SEC           ;SZAM JOTT
630        SBC #'0       ;KIVONJUK "0" ASCCI KODJAT, IGY KONVERTALJUK
635                      ;BINARISSA
640 OSSZEG LDX ALAP      ;ELSO JEGY
650        BNE OCIK      ;NEM, AKKOR ELTOLJUK AZ EDDIGIEKET EGY HELYI-
655                      ;ERTEKKEL
660        STA N         ;ELSO JEGY N-BE
670        LDA #0
680        STA N+1       ;FELSO BYTE 0
690        LDX #4        ;JELOLJUK ALAPBAN, HOGY MAR NEM ELSO JEGYROL VAN SZO
700        STX ALAP
710        JMP CIKLUS    ;ES OLVASSUK A KOVETKEZO KARAKTERT
720 OCIK   ASL N         ;A MAR BEOLVASOTT SZAMRESZT SZOROZZUK 16-TAL
730        ROL N+1       ;AMI EGY HEXA HELYIERTEKNEK FELEL MEG
740        DEX
750        BNE OCIK
760 OSSZF  CLC           ;HOZZAADJUK AZ UJ ERTEKET N-HEZ
770        ADC N
780        STA N
790        BCC 01
800        INC N+1
810 01     JMP CIKLUS    ;ES OLVASSUK A KOVETKEZO KARAKTERT
820 ENDE   TAX           ;VEGJEL X-BE
830        LDA #0        ;NEM VOLT HIBA
840        RTS
850 SYNTAX LDA #1        ;HIBAS VEG ERTEK SZUBRUTINBAN
860        RTS
870        ;* * * * * * * * * * * * * * * * * * * * * * * * *
```

```
 880 HIBA   LDA #KERDO  ;HIBAJELZEST ADUNK KI ES VEGE A DALNAK
 890        JSR CHROUT
 900        LDA #CR
 910        JSR CHROUT
 920        JMP VEGE    ;PROGRAM VEGE
 930        ;* * * * * * * * * * * * * * * * * * * * * * * *
 940 DUMP   LDA #CR     ;SOREMELES
 950        JSR CHROUT
 960 KOVSOR JSR EGYSOR  ;KEZDOCIM NK, VEGCIM N
 970 E1     LDA N+1     ;HA NK>N AKKOR VEGE
 980        CMP NK+1
 990        BMI DUVEG
1000        BNE D1      ;EGYEBKENT MEG EGY SORT KIIRUNK
1010        LDA N
1020        CMP NK
1030        BMI DUVEG
1040        BEQ DUVEG
1050 D1     CLC
1060        LDA NK      ;NK-T MEGNOVELJUK 8-CAL
1070        ADC #8
1080        STA NK
1090        BCC KOVSOR
1100        INC NK+1
1110        JMP KOVSOR
1120 DUVEG  RTS         ;DUMP SZUBRUTIN VEGE
1130        ;* * * * * * * * * * * * * * * * * * * * * * * *
1140 EGYSOR JSR CIMKI   ;EGY DUMP SOR KIIRASA
1150        JSR HEXAKI
1160        JSR ASCIKI
1170        LDA #CR     ;VEGERE SOREMELES
1180        JSR CHROUT
1190        RTS
1200        ;* * * * * * * * * * * * * * * * * * * * * * * *
1210 CIMKI  LDA NK+1    ;SOR KEZDOCIMENEK KIIRASA
1220        JSR BYTEKI  ;FELSO BYTE
1230        LDA NK
1240        JSR BYTEKI  ;ALSO BYTE
1250        LDA #':     ;KETTOSPONT
1260        JSR CHROUT
1270        LDA #SPACE  ;SZOKOZ
1280        JSR CHROUT
1290        RTS
1300        ;* * * * * * * * * * * * * * * * * * * * * * * *
1310 HEXAKI LDY #0      ;A HEXADECIMALIS DUMP KIIRASA
1320 HCIK   LDA (NK),Y  ;KOVETKEZO BYTE
1330        JSR BYTEKI  ;HEXABAN OUTPUTRA
1340        LDA #SPACE  ;PLUSZ EGY SZOKOZ
1350        JSR CHROUT
1360        INY
1370        CPY #8
1380        BMI HCIK    ;HA Y<8 KOVETKEZOT
1390        RTS
1400        ;* * * * * * * * * * * * * * * * * * * * * * * *
```

```
1410 ASCIKI LDY #0        ;A KARAKTERES DUMP KIIRASA
1420 ACIK   LDA (NK),Y    ;KOVETKEZO BYTE
1430        CMP #SPACE    ;HA  < SPACE VAGY
1440        BMI EXTRA
1450        CMP #$7F      ;>127, AKKOR AKARMI LEHET, EZERT -> EXTRA
1460        BEQ AOUT
1470        BPL EXTRA
1480 AOUT   JSR CHROUT    ;EGYEBKENT UGY, AHOGY VAN KIIRJUK
1490        INY
1500        CPY #8
1510        BMI ACIK      ;HA Y<8 AKKOR KOVETKEZOT
1520        RTS
1530 EXTRA  LDA #'.       ;SPECIALIS KARAKTER HELYETT "."-OT IRUNK KI
1540        JMP AOUT
1550        ;* * * * * * * * * * * * * * * * * * * * * * * * *
1560 BYTEKI PHA           ;EGY BYTE-OT HEXABAN KIIR - MEGJEGYEZZUK A VEREMBEN
1570        LSR A         ;ELOSZOR A MAGASABB HELYIERTEKU HEXAJEGYET IRJUK KI
1580        LSR A         ;EZERT ELOSZTJUK 16-TAL
1590        LSR A
1600        LSR A
1610        TAX
1620        LDA HX,X      ;A MEGFELELO KARAKTERT
1630        JSR CHROUT    ;KIIRJUK
1640        PLA           ;EREDETI SZAM VEREMBOL
1650        AND #$0F      ;ALSO JEGY
1660        TAX
1670        LDA HX,X      ;A MEGFELELO KARAKTERT
1680        JSR CHROUT    ;KIIRJUK
1690        RTS
```

Most vegyük sorra a program utasitásait:

160. sor – A programot a $C000-ra helyeztük, mivel a kazetta-puffer számára tul hosszu

230 – 250. sor – Ez a tulajdonképpeni föprogram.
Feladatai a következök:
Beolvas egy értéket (kezdöcim), áttölti NK-ba, majd ha nincs vége még a sornak, beolvas még egy értéket (végcim). Ezután a DUMP szubrutin segitségével kiirja a memóriadumpot.
Az ERTEK beolvasó szubrutin A-ban nem 0 értéket ad vissza hiba esetén, ekkor a HIBA cimkére kell elugrani.
Most lássuk a szubrutinokat!

370 – 440. sor – Az ERTEK szubrutin feladata az inputban soronkövetkezö hexadecimális érték beolvasása a kétbyte-os N változoba. Az elsö ciklus addig olvas, amig szoköztöl különbözö karaktert nem talál. Igy tüntetjük el a bevezetö space-ket. Ha eközben egy CR (sor vége) karakter jön be, az hibának számit, SYNTAX-ra megyünk.

450 - 580. sor - A beolvasó ciklus elsö része. Ez a ciklus nagyon hasonlit az elözöhöz, de itt az értékes jegyek feldolgozása a feladat. A szóköz és a CR lezárja a számot, ekkor ENDE cimkére ugrunk. A hexadecimális jegyként értelmezhetö számjegyek (SZAM) és az A-tól F-ig terjedö betük (BETU) esetén a megfelelö cimkére megyünk, ahol a beérkezett jegyet binárissá konvertáljuk, az eddigi értékhez összegezzük, majd CIKLUS-ra ugrunk vissza. Minden egyéb karakter hibának számit.

590 - 630. sor - A betük ill. a számok konvertálása a megfelelö bináris értékre.

640 - 810. sor - A bejött hexadecimális számjegyet összegezzük az eddig beérkezett számjegyekkel. A gondolatmenet a következö: Az utoljára érkezett hexa jegyet (egyelöre) az egyesek helyiértékén állónak tekintjük. Ha ez volt az elsö számjegy, egyszerüen tároljuk N-ben (660-680. sor), ha viszont már nem az elsö, a helyzet bonyolultabb. Ekkor az eddig beérkezett értéket 4 bináris jeggyel (2^4=16) balra toljuk (720-750. sor), majd hozzáadjuk az uj értéket (760-800. sor). Azt, hogy az elsö számjegyröl van-e szó, ALAP-ban jelöljük. Ennek értéke 0 az elsö jegy feldolgozásánál, késöbb 4-re állitjuk (690-700. sor),ami a jelzön kivül a léptetési ciklus ciklusváltozójául is szolgál. Ezt vizsgáljuk a 640-650. sorokban. A 810. sorban visszaugrunk a CIKLUS-ra, azaz olvassuk az inputból a következö karaktert.

820 - 840. sor - Ez az ERTEK szubrutin logikai befejezése hibátlan esetben. A visszatérési paraméterek: beolvasott végjel X-ben, 0 A-ban.

850 - 860. sor - Hibás esetben viszont A=1-gyel térünk vissza.

880 - 920. sor - A HIBA szubrutin hivása a föprogramból (240. és 330. sor) történik, ha az ERTEK szubrutin hibát jelzett. Itt egy kérdöjelet és egy kocsivissza jelet irunk ki, majd a VEGE cimkére (a program logikai végére) adjuk a vezérlést.

940 -1120. sor - Ez a szubrutin végzi a memóriadump listázását, eddig csak a paraméterek beolvasásával foglalkoztunk. A DUMP ciklusban hivja az EGYSOR szubrutint, ami egy DUMP sor kiiratását végzi az NK által mutatott cimtöl kezdödöen. Minden sor kiirása után megvizsgáljuk, hogy NK értéke nem érte-e el N értékét, ami a DUMP végcime. Ezután megnöveljük NK-t 8-cal és kiirjuk a következö sort az EGYSOR szubrutin segitségével és igy tovább.

1140-1190. sor - Az EGYSOR szubrutin semmi mást nem csinál, mint sorban hivja a sor kezdöcimét kiiró, a dump hexadecimális részét kiiró és a sor ASCII részét kiiró szubrutinokat, majd egy CR-rel lezárja a sort.

1210-1290. sor - Ez a szubrutin az NK aktuális értékét irja ki a BYTEKI szubrutin kétszeri hivásával (felsö byte, alsó byte), majd egy kettöspontot és szóközt ir ki.

1310-1390. sor - A HEXAKI szubrutin egy ciklusban sorraveszi a sorba tartozó 8 byte-ot és a BYTEKI szubrutinnal kiirja értéküket egy-egy szóközzel együtt.

1410-1540. sor - Az ASCIKI nagyon hasonlit az elözöhöz a ciklus szervezésében, hiszen itt is sorra kell venni ugyanazt a 8 byte-ot, de azokat nem a BYTEKI segitségével irjuk ki. Ezek közvetlenül outputra kerülhetnek, kivéve a speciális, pl. formátum- és szinvezérlö karaktereket, amelyek összezavarnák a nyomtatási képet. Az egyszerüség kedvéért (kicsit tul egyszerüsitve a problémát) speciális karakternek tekintünk (EXTRA) minden $20-nól kisebb és $7F-nél nagyobb értékü karaktert. Ezek helyett pont kerül az outputra (1530-1540. sor).

1560-1690. sor - Végül a már többször hivott BYTEKI rutin. Bináris-hexadecimális konvertálást végez ugy, hogy elöször a felsö hexadecimális számjegyet választja le négyszeri jobbra léptetéssel és irja ki, majd az alsó számjegyet logikai AND müvelettel leválasztva konvertálja és küldi outputra. A konvertáláshoz a HX tömböt használja fel, amelyben a K-adik karakter a K értékü hexadecimális számjegy.

Az ismertetett program talán egy kicsit sok szubrutint tartalmaz, ami nem a leggyorsabban futó kódot eredményezi, viszont igen hasznos mind a program megértése, mind tesztelése és esetleges módositása szempontjából.

5.3 A Commodore 64 "operációs rendszere", a KERNAL

Mint már esett szó róla, a tár $E000-tól $FFFF-ig terjedö területén a gép alapvetö programjai, a KERNAL szubrutin-gyujteménye található. Számunkra ez ugy jelentkezik, mint a programozáshoz segitséget adó eszköz, ezért érdemes megismerkedni **a legfontosabb szubrutinok interface**-ével és rentésével. A felhasználó által hivható valamennyi szubrutin interface-e megtalálható a függelékben.

CHROUT - egy byte-ot ir ki az éppen aktiv output file-ra. Ha nem volt még ilyen megnyitva (CHKOUT--tal), az output a képernyöre megy.
Belépési cim: $FFD2
Paraméterek : A -kinyomtatandó karakter ASCII kódja
Példa : LDA #$0D
JSR CHROUT
egy kocsivissza-jelet ir ki

CHRIN - egy byte-ot olvas az éppen aktiv input file-ról. Ha nem volt még ilyen megnyitva (CHKIN-nel), a billentyüzetröl olvas. Utóbbi esetben a képernyöszerkesztöt aktivizálja, amely villogtatja az aktuális pozicióban a cursort, beolvas egy logikai sort a BASIC-sorpufferbe ($200), majd mindenegyes CHRIN hivásra egy karaktert ad át, egészen addig, amig a puffer ki nem ürül, ekkor az egész elölröl kezdödik.
Belépési cim: $FFCF
Paraméterek : A - a beolvasott karakter ASCII kódja
Példa : JSR CHRIN
STA BYTE

CHKIN - megnyit olvasásra egy file-t. A file-sorszámot az X regiszterben várja.
Belépési cim: $FFC6
Paraméterek : X - file-sorszám
Példa : LDX #5
JSR CHKIN
elökésziti olvasásra a korábban megnyitott 5-ös file-t

CHKOUT - megnyit irásra egy file-t. A file-sorszámot az X regiszterben várja.
Belépési cim: $FFC9
Paraméterek : X - file-sorszám
Példa : LDX #5
JSR CHKOUT

CLRCHN - lezárja a nyitott input és/vagy output file-okat, de a file-ok nyitva maradnak.
Belépési cim: $FFCC
Paraméterek : -

A fenti szubrutinok használatával igen egyszerüen valósitható meg az assembly programok input/output tevékenysége. A file-ok megnyitását és lezárását BASIC-ből vagy más KERNAL-rutinokkal végezhetjük.

A gép ROM-rutinjai ugy lettek tervezve, hogy a felhasználó könnyen beavatkozhasson müködésükbe. Igy pl. a felhasználó által is hivható KERNAL-rutinok többsége a memória RAM részében elhelyezett vektorokon keresztül léptethető be. Ezen vektorok átirásával a KERNAL-rutin meghivásakor a vezérlést saját programunkhoz terelhetjük és ott tetszés szerint ténykedhetünk. Igaz ez a BASIC-interpreter néhány fontosabb szubrutinjára is. Az átirható KERNAL-vektorok:

```
$31A-$31B  -  OPEN
$31C-$31D  -  CLOSE
$31E-$31F  -  CHKIN
$320-$321  -  CHKOUT
$322-$323  -  CLRCHN
$324-$325  -  CHRIN
$326-$327  -  CHROUT
$328-$329  -  STOP
$32A-$32B  -  GETIN
$32C-$32D  -  CLALL
```

Lássunk egy példát a KERNAL átirására! Tegyük fel, hogy egy speciális printerünk van, amely bizonyos kódokat másként értelmez, mint a Commodore 64 belső kódrendszere, ami egyébként csak nagyjából felel meg az ASCII-nak. Igy lehet ez például egy olyan nyomtató esetében, amely ékezetes betüket tud nyomtatni, de ezek a betük nem azon karakterek kódjainak felelnek meg, amelyek lenyomásával mi rögziteni kivánjuk őket, pl. a magyar irógépszabvány billentyüi szerint. A képernyokódok megfelelő összerendelését a karakterképekkel könnyü megoldani, de azután még el kell végezni az átkódolást minden egyes karakter esetén, amelyet a nyomtatóra külküldünk. Az alábbi kis szubrutin első belépési pontja a CHROUT-vektort irja át saját rutinunkra, amely az átkódolást végzi:

```
10  CHROUV = $326
20  CHROUT = $F1CA
30  EGYSEG=$9A
40  *=$CE00
50  BEALL   LDA #SAJAT<
60          STA CHROUV
```

```
 70          LDA #SAJAT>
 80          STA CHROUV+1
 90          RTS
100  SAJAT   PHA            ; OUTPUT BYTE STACK-BE
110          LDA EGYSEG
120          CMP #4         ; PRINTER?
130          PLA ATKOD
140          TXA            ; EREDETI BYTE
150          JMP CHROUT     ; OUTPUTRA
160  ATKOD   LDA TABLA,X    ; MEGFELELO KOD
170          JMP CHROUT     ; OUTPUTRA
180  TABLA   .BYTE 0,1,2,3,4,5,6,7,8,9,$A,$B,$C,$D,$E,$F
...          ...
```

Még egy megjegyzés a fenti programhoz: A $9A cim az aktuális output eszköz egységszámát tartalmazza.

5.4 BASIC és assembly békés egymás mellett élése

A Commodore 64 gépre készülö assembly programok többsége a BASIC-interpreterrel és a BASIC-programokkal igen szoros kapcsolatot tart. Leggyakrabban olyan BASIC-kiegészitéseket programozunk assembly nyelven, amelyek a BASIC programból nehezen vagy nem elég hatékonyan megoldható feladatokat teljesitenek.

A BASIC-kiegészitések programozásához természetesen alaposan ismerni kell a ROM-ban lévö interpreter mükódését, memóriahasználatát, belépési pontjait, a BASIC program felépitését, stb. Mindezen ismereteket itt nem lenne helyénvaló és nem is szükséges felsorolni, ezek nagyrészét a Commodore BASIC-kel foglalkozó irodalom tartalmazza, ld. pl. (1), (3).
Ebben a fejezetben csak azokat az ismereteket foglaljuk össze, amelyek speciálisan az assembly nyelvü kiegészitéseket érintik és más munkákban nem,vagy nem ilyen szempontból vannak leirva. **Feltételezzük egy sor olyan ismeret meglétét, amely inkább a BASIC programozáshoz tartozik egészen a BASIC tárfelosztás és interpreter-kód mélységéig.** Ha az olvasó nem rendelkezik ezen ismeretekkel, vagy lapozza át ezt a fejezetet, vagy tegyen szert ezekre az ismeretekre pl. a fent megjelölt munkákból, és csak azután tanulmányozza ezt a fejezetet.

Az egész változók (jele %) egyedi változóként 7, tömbelemként 2 byte-ot foglalnak a memoriában, de az elso esetben a 7-böl 3 byte kihasználatlan. Két byte-on a változo neve és tipusa, két byte-on maga az értéke van elhelyezve nem cim formában, hanem **felsö byte - alsó byte sorrendben.**
Az egyedi változó tehát:

0.	byte	-	név elso karaktere + 128
1.	byte	-	név második karaktere vagy 0+128
2.	byte	-	magasabb helyiértékü (F) byte
3.	byte	-	alacsonyabb helyiértékü (A) byte
4-6.	byte	-	közömbös (0)

Az egész tömb szerkezete:

0. byte	-	név elsö karaktere + 128
1. byte	-	név második karaktere vagy 0+128
2-3. byte	-	következö tömb kezdöcime, a tömb kezdetéhez relativ mutató!
4. byte	-	dimenziók száma (n)
5-tol 2*n+4-ig	-	az egyes dimenziók felsö határai két-byte-os egész formában (azaz felsö - alsó byte sorrendben).
elemszám*2 byte	-	a tömb értékei, mint az egyszerü változónál (felsö-alsó byte) sorrendben.

A lebegöpontos numerikus változók számára a BASIC 2+5=7 byte-ot tart fenn, a tömbelemek számára 5-öt. A FAC - lebegöpontos akkumulátor - tárgyalásánál megtudtuk, hogy a szám ábrázolásához 5 byte-ra, valamint az elöjel tárolására egy külön byte-ra van szüksége az interpreternek. A változók számára viszont elég 5 byte. **Hogy is van ez?**

A lebegöpontos(FAC formátumu) ábrázolásban elöször is a számot binárissá kell alakitani, majd ugy beirni a mantissza tárolására szolgáló 1-4. byte-okba, hogy a szám elsö 1-ese a legelsö pozicióra kerüljön. Pl. a 12 binárisan:

1100

Igy kerül a FAC-ba:

	0.	1.	2.	3.	4.
decimálisan :	132	102	0	0	0
hexadecimálisan:	84	C0	0	0	0
binárisan :	1000 0100	1100 0000	0	0	0

A 0. byte-ot, az exponenst ugy állitjuk elö, hogy az elsö értékes jegy elöjeles távolságát a bináris ponttól (esetünkben 4) hozzáadjuk 128-hoz. A szám elöjele megegyezik a FAC 5. byte-jának elöjelével. A nulla kódja megegyezés szerint csupa nulla byte. A memóriaformátum már egy kicsit különbözik a FAC formátumtól, mivel annál nem tartunk fenn egy külön byte-ot az elöjelnek. A fenti algoritmusból következöen a FAC formáju szám 1. byte-jának 0. bitje mindig 1, kivéve, ha értéke nulla. Igy ez a bit nem hordoz információt, nyugodtan helyettesithetjük ezt a bitet a szám elöjelbitjével (1, ha negativ, 0, ha pozitiv). A memóriában tárolt lebegöpontos érték igy különbözik egy kicsit a FAC-ban számolásra kész értéktöl, töltéskor ill. tároláskor a konverzióról az interpreternek gondoskodni kell. A változók és tömbök szerkezete megegyezik az egész változók és tömbökével, csak az érték hossza mindenütt 5 byte, valamint a név változatlanul kerül tárolásra, ezzel jelezve a tipust.

A karakteres változók már nem tartalmazzák a változó értékét, mivel a stringeket a BASIC interpreter egy külön összefüggö területen helyezi el a változó hosszuságu string-kezeléshez szükséges szemétgyüjtés megkönnyitésére. A karakteres változó ill. tömbelem értékes része 3 byte hosszu, amelyböl a 0. byte a karakterek száma, az 1-2. byte-ok pedig a string elsö karakterének az abszolut cime a memóriában.

A fenti számábrázolási formátumok egymásba konvertálása, a kifejezések kiértékelése, a változók és tömbelemek megkeresése, uj változók deklarálása, létrehozása bonyolult prog-

ramozási feladatot jelentene, de szerencsére **nem kell megcsinálni, mivel ezt megtették már az interpreter irásakor.** Miért is ne használnánk ezeket a szubrutinokat assembly programunkból? Az alábbiakban az interpreter olyan belépési pontjait - és azok paraméterezését - ismertetjük, amelyek talán leghasznosabbak a BASIC-kiegészitések irásakor:

FRMEVL - A soronkövetkező BASIC kifejezés kiértékelése és az eredmény elhelyezése FAC-ban. Ha a kifejezés karakteres volt, FAC 3-4. byte-ja ($64-$65) a string 3 byte-os leirójára mutat.
Belépési pont: $AD9E

INTFAC - Az A (alsó byte) és az Y (felső byte) regiszterekben lévő kétbyte-os előjeles egész szám konvertálása lebegőpontos formára és elhelyezése FAC-ban.
Belépési pont: $B391

YTOFAC - Az Y regiszter tartalmának konvertálása lebegőpontos formára és elhelyezése FAC-ban.
Belépési pont: $B3A2

MEMFAC - Az A (alsó byte) és Y (felső byte) egy memóriacimet tartalmaz. A cimen lévő 5 byte-os lebegopontos szám átvitele FAC-ba (formátumváltással).
Belépési pont: $BBA2

STRFAC - String konvertálása lebegopontos formára és az érték elhelyezése FAC-ban. A string hosszát az A regiszter, kezdocimet $22-$23 tartalmazza.
Belépési pont: $B7B5

MEMFC2 - Ugyanaz, mint MEMFAC, de az eredmény a második lebegopontos akkumulátorba,FAC2-be kerül ($69-$6E). A $6F cimen lévő byte 0. bitje 1 lesz, ha FAC és FAC2 előjele különböző. Ez a byte az ún. elojel-összehasonlitási jelző (SCF - sign comparasion flag).
Belépési pont: $BABC

FACINT - FAC tartalmát kétbyte-os elojeles egésszé konvertálva $64-$65-be teszi. FAC-ot elrontja. Az egész szám felső byte - alsó byte formájú.
Belépési pont: $B1AA

FACADR - Ugyanaz, mint FACINT,de az eredmény kétbyte-os elöjeltelen egész (0-65535) cim formátumban (alsó byte - felső byte) a $14-$15 cimre kerül. FAC-ot elrontja.
Belépési pont: $B7F7

FACMEM - MEMFAC forditottja. A memóriaterület címét X (alsó byte) és Y (felsö byte) regiszterekben várja el a rutin.
Belépési pont: $BBD7

FACSTR - A FAC tartalma karaktersorozattá konvertálva a $100 cimre kerül. A stringet egy 0 byte zárja.
Belépési pont: $BDDD

FC2FAC - FAC2 másolása FAC-ba.
Belépési pont: $BBFC

FACFC2 - FAC másolása FAC2-be.
Belépési pont: $BC0F

FPLUS - FAC=FAC+FAC2
A rutin hivása elött az SCF-et megfelelöen állitani kell (LDA $66 / EOR $6E / STA $6F) és az A regiszterbe a FAC exponensét kell tölteni (LDA $61).
Belépési pont: $B86A

FMINUS - FAC=FAC-FAC2
Belépési pont: $B853

FMULT - FAC=FAC*FAC2
Az elökészités ugyanabból áll, mint FPLUS-nál.
Belépési pont: $BA30

FDIV - FAC=FAC2/FAC
Elökészités, mint FPLUS-nál.
Belépési pont: $BB12

FPOWER - FAC=FAC2^FAC
Elökészités, mint FPLUS-nál.
Belépési pont: $BF7B

ADRESS - A BASIC szövegben következö változó cimének elökeresése (lehet tömbelem is). A változó cimét (ahol a neve megtalálható) a $5F-$60-ba irja. Tömbelem megadása esetén ez a tömb nevére (kezdetére) és $59-$5A a tömb elsö értékére mutat. Az A (alsó byte) és az Y (felsö byte) regiszterek magára az értékre (string esetén a a hárombyte-os leiróra) mutatnak. Ugyanezt a cimet tartalmazza $47-$48 is.
$0D egyenlö $FF-fel, ha a változó karakteres tipusu, egyébként 0 (string flag). $0E egyenlö $80-nal, ha a változo egész tipusu, egyébként 0 (integer flag).
Belépési pont: $B08B

LINEIN – Egy logikai sor beolvasása a képernyöszerkesztö segitségével a BASIC input pufferbe ($200). A sor maximális hossza 80 byte, a karaktersorozatot bináris egy 0 byte zárja le.
Belépési pont: $A560

LINOUT – Egy karaktersorozat kiirása a képernyöre az aktuális poziciótól kezdve. Az A (alsó byte) és Y (felsö byte) regiszterekben várja a szubrutin a karaktersorozat kezdöcimét.
Belépési pont: $AB1E

Lássunk egy példát, amelyik a fenti szubrutinok közül talán a legbonyolultabbat, az ADRESS-t használja. A program feladata a más BASIC-interpreterek által elfogadott ERASE utasitás megvalositása.

```
 10  ; ERASE SZUBRUTIN TOMB TORLESEHEZ
 20  ; HIVASA SYS 49152,A$(0,0...), AHOL AZ INDEXEK
 30  ; ERTEKE TETSZOLEGES. A TOMBNEK MAR DEKLARALTNAK
 40  ; KELL LENNI. A SZUBRUTIN LEFUTASA UTAN A TOMB
 50  ; UJRA DIMENZIONALHATO
 60  ADRESS=$B08B
 70  TVEGE =$31
 80  CHRGET=$73
 90  TCIM  =$5F
100  KOV   =$47
110  *     =$C000
120  ERASE    JSR  CHRGET          ; TERMINATOR
130           JSR  ADRESS          ; TOMB CIME TCIMBE
140           LDY  #2              ; KOVETKEZO TOMB CIME
150           LDA  (TCIM),Y
155           ADC  TCIM
160           STA  KOV             ; KOV-BE
170           INY
180           LDA  (TCIM),Y
185           ADC  TCIM+1
190           STA  KOV+1
200           LDY  #0
210  HASONL   LDA  KOV             ; KOV=TVEGE?
220           CMP  TVEGE
230           BNE  CIKLUS
240           LDA  KOV+1
250           CMP  TVEGE+1
260           BEQ  CIKVEG          ; IGEN ---> VEGERE
270  CIKLUS   LDA  (KOV),Y         ; EGY BYTE MASOLASA
280           STA  (TCIM),Y
290           INC  KOV             ; MUTATOK INKREMENTALASA
300           BNE  F1
310           INC  KOV+1
320  F1       INC  TCIM
330           BNE  HASONL
340           INC  TCIM+1
350           JMP  HASONL
```

```
360 CIKVEG  LDA  TCIM      ; TOMBVEG-POINTER CSOKKENTESE
370         STA  TVEGE
380         LDA  TCIM+1
390         STA  TVEGE+1
400         RTS            ; VISSZA A BASIC-HEZ
```

Nézzük lépésről-lépésre!

120. sor – Az elválasztó vessző átlépése.

130. sor – A tömböt megkeresi a BASIC-területen és kezdőcimét $5F-be (TCIM) teszi.

140-190. sor – A tömb leirásából kiemeljük a következő tömb cimét (2.-3. byte) és KOV-be rakjuk. A feladat tulajdonképpen az ettől a cimtől a BASIC tömbök végéig (TVEGE=$31-$32) terjedő terület visszacsusztatása TCIM-re. Ezzel eltüntetjük a paraméterként kapott tömböt.

200. sor – Y regiszter nullázása indexeléshez.

210-260. sor – Megvizsgáljuk, hogy KOV értéke elérte-e már TVEGE értékét. Ha igen, vége a másolásnak.

270-280. sor – Egy byte másolása.

290-350. sor – KOV és TCIM növelése és visszatérés a ciklus elejére (HASONL).

360-400. sor – TCIM most oda mutat – pontosabban az utolsó utáni byte-ra – ahova a tömbök hátralévő részét átmásoltuk. Ez a cim lesz a TVEGE – mutató uj értéke. Ezután visszatérünk a BASIC-hez.

A BASIC-kel való kapcsolattartás lényeges része a BASIC program és az assembly szubrutin vezérlésátadásának és paraméterezésének kérdése.

Már megismerkedtünk ezen módszerek egy részével, de azért foglaljuk a legfontosabbakat össze:

Assembly rutin hivása BASIC-ből:

- SYS BASIC utasitással,
- USR függvényként,
- beszurásként.

Az utóbbiról még nem volt szó, ez tulajdonképpen a BASIC-interpreter módositása, kiegészitése. Mint a KERNAL-rutinok némelyike, a BASIC-interpreter néhány rutinja is ugy lett tervezve, hogy **beléptetésük egy RAM-ban tárolt vektoron át történik**, amelynek átirásával a ROM-ban lévö szubrutin helyett saját programunkhoz kerül a vezérlés. Ezután természetesen visszatérhetünk az interpreter bármely pontjára, akár a kiváltott BASIC-interpreter rutinra is. A RAM-ban tárolt vektorok és a hozzájuk tartozó szubrutinok:

$300-$301	-	Hibaüzenet kiirása
$302-$303	-	BASIC melegindités
$304-$305	-	Egy sor átváltása interpreter kódra (tokenizálás)
$306-$307	-	Interpreter kód átváltása LIST tipusu sorrá
$308-$309	-	A következö BASIC utasitás végrehajtása
$30A-$30B	-	Numerikus term (konstans vagy változó) bevitele FAC-ba. FRMEVL hivja a kifejezés minden tagjánál (nemigen érdemes átirni).

A beszurások onnan kapták nevüket, hogy a BASIC-interpreter kódvégrehajtási folyamatába szurjuk be azokat és segitségükkel lehetöségünk nyilik annak eldöntésére, hogy nekünk, vagy az interpreternek szóló parancsról van-e szó. Leggyakoribb a $308-$309 vektor átirása. Ha ezt saját szubrutinunk belépési pontjára állitjuk át,minden BASIC utasitás elött nekünk adódik a vezérlés. Ekkor $7A-$7B a soronkövetkezö karakterre mutat a BASIC program szövegében. Ha az utasitás vizsgálata azt az eredményt adja, hogy nekünk szól, hajtsuk végre és gondoskodjunk arról, hogy $7A-$7B-t a következö utasitás elejére állitsuk. Ellenkezö esetben egyszerüen ugorjunk JMP-pal a $A7E4 cimre, ami a $308-$309 alapértéke. Ha az interpreter-kódra váltó rutin elé is beszurást teszünk (a $304-$305 vektor átirásával), lehetöségünk nyilik arra, hogy saját parancsainkat is felismerjük és egybyte-os kódra - tokenre cseréljük. Ekkor az utasitás végrehajtásnál lényegesen felgyorsul, egyetlen byte ellenörzésére csökken a beszurás feladata. Ilyenkor persze nemcsak az interpreter-kódra váltó, hanem az arról olvasható szövegre konvertáló (vektor: $306-$307) rutin elé is beszurást illik tenni, hogy a LIST olvasható programot irjon ki.

Paraméterátadás BASIC-böl assembly-be:

- **SYS utasitással hivott szubrutinnak** a már ismert módon, a FRMEVL interpreter rutin segitségével adhatunk át tetszés szerinti számu és tipusu paramétert, kifejezéseket is. Másik megoldás a regiszterek használata. Ha BASIC programból SYS-szel hivunk egy gépikódu cimet, az interpreter a rutin meghivása elött az A,X,Y,P regisztereket rendre a $30C,$30D,$30E,$30F byte-ok tartalmával tölti fel, majd a rutin visszatérése után a regisztereket ide menti el. Ezeket a memóriacimeket a BASIC program PEEK-kel és POKE-kal elérheti, igy ezeken keresztül egyszerü paraméterátadás valósitható meg.

- **USR-rel hivott rutin** egy numerikus értéket kap a FAC-ban és ugyanott egy numerikus értéket ad is vissza. Ezenkivül csak memórián keresztül adható át paraméter.

- **BASIC-beszurás** esetén rajtunk áll, hogy értelmezzük a nekünk szóló utasitást és az interpreter mely részeit használjuk fel az utasitás végrahajtása során.

Visszatérés a BASIC-interpreterbe:

mindhárom tipusu assembly-kiegészités **RTS-sel** kell, hogy végetérjen, hibás esetektöl eltekintve, amikor a megfelelö hibaágra kell ugranunk.

5.5 Gyorsabban, messzebbre, magasabbra...

A sportból kölcsönzött fejezetcim jól jelképezi azt, hogy elsösörban milyen tipusu feladatok megoldására érdemes használni az assembly programozás lehetöségeit. Igen fontos, szinte elkerülhetetlen alkalmazása azokban az esetekben, amikor a magas szintü nyelven - általában BASIC-ben - irt megoldáshoz képest a végrehajtási idö jelentös felgyorsitása a cél, vagy a probléma természetéböl adódóan, vagy egyszerü ergonómiai szempontok alapján. Az elsöre példa szinte **valamennyi komolyabb játékprogram**, ahol a grafika animációja, a hangeffektusok kezelése, a felhasználói beavatkozásokra történö reagálás olyan végrehajtási sebességet követel meg, amely BASIC-böl, de jószerivel a többi magas szintü nyelvböl sem biztositható. Az utóbbira inkább az adatfeldolgozás köréböl hozhatunk fel példákat. Magasszintü nyelven megirva bizonyos elemi adatkezelési tevékenységek (**rendezés, válogatás, keresés, összefuttatás**) annyira lassuak, hogy a felhasználótól nem követelhetö meg, hogy kivárja azokat. Különösen igaz ez az **adatbáziskezelésre**, magasszintü file-kezelésre, ill. ezek visszakeresö és file-módositó funkcióira.

Mivel az elsö tipusu feladatok (játék-software) inkább egyedi, speciális problémákat vetnek fel, inkább az utóbbi területröl választunk szemléltetö példát. Emellett szól az is, hogy mig egy játékprogram valamely érdekesebb részének elemzése nem nyujtana kis változtatással felhasználható épitökockát olvasóink további programozási munkájához, addig az itt következö programok részleteinek vagy egészének alkalmazása véleményunk szerint hasznos lehet.

Az egyik legidöigényesebb adatfeldolgozási müvelet a **rendezés**. Anélkül, hogy a rendezési algoritmusok elméleti kérdéseibe tulságosan belemennénk, bevezetöben elengedhetetlen néhány szót szólni a rendezés módszereiröl.

Erre a feladatra igen sok algoritmus ismeretes. Itt természetesen csak **a legegyszerübb algoritmusok** bemutatásáról lehet szó, de ezek a mikrogépes feladatok legfeljebb néhány ezer rekord rendezési igényének tökéletesen megfelelnek, amellett gyorsan és egyszerüen programozhatók. A legegyszerübb még használható algoritmus **a buborék-módszer.**
Ennek során minden egyes cikluslépésben kikeressük a még rendezetlen lista legnagyobb elemét s egyszersmind a legutolsó pozicióba juttatjuk.

Ezt ugy végezzük, hogy az egymás után következö elemeket páronként összehasonlitjuk egymással. Ha a második elem kisebb az elötteállónál, felcseréljük öket. Végig folytatva a legnagyobb elem az utolsó helyre vándorol, tehát elegendö a következo ciklust az öt megelözö elemig folytatni, (az már eggyel rövidebb lesz), amig csak el nem érünk az elsö, most már biztosan legkisebb elemhez. A módszer onnan kapta a nevét, hogy a kisebb elemek a rendezés folyamán ugy szállnak

feljebb és feljebb, mint a légbuborékok a folyadékban. A hatékonyságot egy egyszerü trükkel fokozhatjuk, ha egy jelzöt használunk arra, hogy egy ciklusban történt-e elemcsere,vagy sem. Ha nem történt, nyugodtan befejezettnek nyilvánithatjuk a rendezést, hiszen ez azt jelenti, hogy minden elem kisebb-egyenlö az utánakövetkezönél, tehát a lista rendezett. Erre a javitott algoritmusra közlünk egy programot, amely egy egész tipusu BASIC-tömb rendezését végzi, természetesen helyben:

```
 10  ; BUBOREK MODSZER EGESZ TOMB RENDEZESERE
 20  ; AZ ELEMEKET ELOJEL NELKULI EGESZ SZAMOKNAK TEKINTJUK
 30  ;
 40  ; HIVASA : SYS49152,R%(0)  , AHOL R% A RENDEZNI KIVANT TOI
 50  ;
 55  CHRGET=$73
 60  MERET = $FB
 70  K     = $FD
 80  AKT   = $5A
 90  KOV   = $5C
100  JELZO = $FF
110  TCIM  = $5F
120  ADRESS= $B08B
125       *= $C000
127          JSR CHRGET      ; VESSZO
130          JSR ADRESS      ; TOMB KEZDOCIM TCIM-BE
140          LDY #5          ; TOMB MERETE
150          LDA (TCIM),Y
160          STA MERET+1     ; MERET-BE
170          INY
180          LDA (TCIM),Y
190          STA MERET
200  KULSO   LDA TCIM        ; KOV=TCIM+7 --> 0. ELEM CIME
210          CLC
220          ADC #7
230          STA KOV
240          LDA TCIM+1
250          ADC #0
260          STA KOV+1
270          LDA #0
280          STA K           ; CIKLUSVALTOZO NULLAZASA
290          STA K+1
300          STA JELZO       ; CSERE-JELZO LEGYEN HAMIS
310  BELSO   LDA KOV         ; AKT=KOV : KOV=KOV+2
320          STA AKT
330          CLC
340          ADC #2
350          STA KOV
360          LDA KOV+1
370          STA AKT+1
380          BCC U2
390          INC KOV+1
400  U2      LDY #1          ; OSSZEHASONLITAS
410          SEC
420          LDA (KOV),Y
```

```
430            SBC (AKT),Y
440            DEY
450            LDA (KOV),Y
460            SBC (AKT),Y
470            BCS NEMCSE         ; NAGYOBB
480            LDX #1             ; VOLT CSERE
490            STX JELZO
500 CSERE      LDA (AKT),Y
510            TAX
520            LDA (KOV),Y
530            STA (AKT),Y
540            TXA
550            STA (KOV),Y
560            INY
570            CPY #2
590            BNE CSERE
600 NEMCSE     INC K              ; K=K+1
610            BNE U3
620            INC K+1
630 U3         LDA K              ; HA K=MERET, VEGE
640            CMP MERET          ; A BELSO CIKLUSNAK
650            BNE BELSO
660            LDA K+1
670            CMP MERET+1
680            BNE BELSO
690            LDA JELZO          ; VOLT CSERE?
700            BEQ VEGE           ; NEM
710            LDA MERET          ; MERET=MERET-1
720            SEC
730            SBC #1
740            STA MERET
750            BCS U4
760            DEC MERET+1
770 U4         LDA MERET          ; HA MERET=0, VEGE
780            BNE KULSO
790            LDA MERET+1
800            BNE KULSO
810 VEGE       RTS                ; VISSZA A BASIC-HEZ
```

Nézzük a programot sorról-sorra!

130. sor – A tömbnév paraméter átvétele BASIC-ból és a tömbcim kiszámitása az ADRESS interpreter-rutin segitségével.

140–190. sor – A tömb deklarált méretének kiemelése a tömb leirójából és a MERET változóba irása.

200–260. sor – A külső ciklus feje. KOV mutatót a tömb 0. elemére állitjuk.

270-300. sor - Nullázzuk a csere-jelzöt és a ciklusváltozót, amely a belsö ciklus hosszáig nö majd.

310-390. sor - A belsö ciklus feje. Az AKT mutatót a KOV-vel tesszük egyenlövé, a KOV az AKT utáni elemre állitódik.

400-470. sor - Az egymást követö két szám kivonásával eldöntjük, melyik nagyobb. Ha a második, nincs csere.

480-490. sor - Ha az elsö, a jelzöben jelöljük a cserét, majd

500-590. sor - megcseréljük a két elemet.

600-620. sor - Eggyel növeljük a ciklusváltozó értékét.

630-680. sor - Ha kisebb, mint az aktuális ciklusméret, folytatjuk a vizsgálatot,

690-700. sor - ha elértük a ciklus végét, megvizsgáljuk, volt-e csere. Ha nem, kész a rendezés.

710-800. sor - Eggyel csökkentjük a MERET-et, s ha nullára ért, véget vetünk a rendezésnek, egyébként folytatjuk a külsö ciklust.

810. sor - Vissza a BASIC-hez, a tömb rendezett.

A fenti módszer nagyon jól használható **nem tul nagy méretü** tömbök rendezéséhez. Ha azonban a méret növekedni kezd, és különösen ha nem egész számok, hanem karaktersorozatok rendezését kell végezni vagy beletörödünk abba, hogy néhány száz elem rendezése már perceket igényel, vagy uj módszer után nézünk.

Ha nem akarjuk teljesen eldobni a buborék-rendezés programját és a vele szerzett tapasztalatokat, érdemes egy olyan eljárást keresni, amelyik annak továbbfejlesztése. A buborék-rendezésröl már tudjuk, hogy akkor hatékony, ha aránylag kevés elemet kell rendezni és komolyan felgyorsitható, ha sejtjük, hogy a lista közel rendezett és alkalmazzuk a csere jelzöt. Ezekböl az információkból már kinálkozik a **keverö-rendezés (merge sort)** módszere, ami abból áll, hogy az elemeket sok kis halmazra osztjuk, ezeket buborék-módszerrel rendezzük, a kis halmazokat nagyobb, részben rendezett listákká futtatjuk össze, ezeket buborék-rendezzük s.i.t, mig el nem érjük a teljes tömb méretét.

Hogy a példaprogram ne hasonlitson olyan nagyon az elözöre, ezt a módszert egy **karakteres BASIC tömb helybenrendezésén** mutatjuk be:

```
100      ;      MERGE  S/ASS
110      ; HIVASA BASIC-BOL: SYS 50688,A$(X),N AHOL A$(X) A
115      ; KEZDOELEM, N AZ ELEMEK SZAMA
120 MERET  =$FB
130 K      =$FD
140 AKT    =$57
150 KOV    =$59
160 JELZO  =$FF
170 STR1   =$5B
180 STR2   =$5D
190 HOSSZ1 =$5F
200 HOSSZ2 =$60
210 KITEVO =$4E
220 NAGYSG =$4F
230 L      =$51
240 TAROL  =$26
250 CHRGET =$73
260 FRMEVL =$AD9E
270 ADRESS =$B08B
280 GETADR =$B7F7
290 TCIM   =$47
300 Z1     =$14
310        *=50688
320        JSR CHRGET ; TERMINATOR BE
330        JSR ADRESS ; AZ ELSO RENDEZENDO TOMBELEM CIME A KCIMBE
332        LDA TCIM   ; TCIM ELMENTESE KCIM-BE
334        STA KCIM
336        LDA TCIM+1
338        STA KCIM+1
340        JSR CHRGET       ; MASODIK VESSZO
350        JSR FRMEVL       ; MERET FAC-BA
360        JSR GETADR       ; CIM FORMABAN Z1-BE
370        LDA Z1           ; MERET-BE
380        STA MERET
390        LDA Z1+1
400        STA MERET+1
410        LDA #1           ; NAGYSG=1
420        STA NAGYSG
430        LDA #0           ; KITEVO=0
440        STA KITEVO
450        STA NAGYSG+1
460 HATVC  INC KITEVO       ; KITEVO=KITEVO+1
470        ASL NAGYSG       ; NAGYSG=2*NAGYSG
480        ROL NAGYSG+1
490        SEC
500        LDA NAGYSG       ; IF NAGYSG<MERET THEN HATVC
510        SBC MERET
520        LDA NAGYSG+1
530        SBC MERET+1
540        BCC HATVC
550 KULSO  LSR NAGYSG+1     ; NAGYSG=NAGYSG/2
560        ROR NAGYSG
570        SEC
580        LDA MERET        ; K=MERET-NAGYSG
590        SBC NAGYSG
600        STA K
```

```
 610          LDA MERET+1
 620          SBC NAGYSG+1
 630          STA K+1
 640 KOZEPS   LDA #0          ; JELZO=0
 650          STA JELZO
 660          STA L           : L=0
 670          STA L+1
 680          LDA KCIM        ; AKT=KCIM
 690          STA AKT
 700          LDA KCIM+1
 710          STA AKT+1
 720          LDA NAGYSG+1    ; TAROL=NAGYSAG
 730          STA TAROL
 740          LDA NAGYSG      ; KOV=3*NAGYSG+AKT
 750          ASL A
 760          ROL TAROL
 770          CLC
 780          ADC NAGYSG
 790          ADC AKT
 800          STA KOV
 810          LDA TAROL
 820          ADC NAGYSG+1
 830          ADC AKT+1
 840          STA KOV+1
 850 BELSO    LDY #0          ; HOSSZ1 ES HOSSZ2
 860          LDA (AKT),Y
 870          STA HOSSZ1
 880          LDA (KOV),Y
 890          STA HOSSZ2
 900          INY
 910          LDA (AKT),Y     ; STR1 ES STR2
 920          STA STR1
 930          LDA (KOV),Y
 940          STA STR2
 950          INY
 960          LDA (AKT),Y
 970          STA STR1+1
 980          LDA (KOV),Y
 990          STA STR2+1
1000          LDY #0
1010 HASON    LDA (STR2),Y    ; IF (STR2)<(STR1) THEN CSERE
1015                          ; ELSE NEMCSE
1020          CMP (STR1),Y
1030          BCC CSERE
1040          BNE NEMCSE
1050          INY
1060          CPY HOSSZ1
1070          BEQ NEMCSE
1080          CPY HOSSZ2
1090          BEQ CSERE
1100          BNE HASON
1110 HOPP1    BNE KOZEPS      ; VISSZAUGRASOK
1120 HOPP2    BNE KULSO
1130 CSERE    LDY #2          ; VOLT CSERE
1140          STY JELZO
1150 CSEREC   LDA (AKT),Y     ; CSERECIKLUS
```

```
1160        TAX
1170        LDA (KOV),Y
1180        STA (AKT),Y
1190        TXA
1200        STA (KOV),Y
1210        DEY
1220        BPL CSEREC
1230 NEMCSE INC L              ; L=L+1
1240        BNE U1
1250        INC L+1
1260 U1     LDA AKT            ; AKT=AKT+3 -> KOVETKEZO ELEM
1270        CLC
1280        ADC #3
1290        STA AKT
1300        BCC U2
1310        INC AKT+1
1320 U2     LDA KOV            ; KOV=KOV+3 -> KOVETKEZO ELEM
1330        CLC
1340        ADC #3
1350        STA KOV
1360        BCC U3
1370        INC KOV+1
1380 U3     LDA K              ; IF L<K THEN GOTO BELSO
1390        CMP L
1400        BNE BELSO
1410        LDA K+1
1420        CMP L+1
1430        BNE BELSO
1440        LDA JELZO          ; IF NOT JELZO THEN GOTO JELZOH
1450        BEQ JELZOH
1460        SEC
1470        LDA K              ; K=K-1
1480        SBC #1
1490        STA K
1500        BCS U4
1510        DEC K+1
1520        LDA K
1530 U4     BNE HOPP1          ; IF K<>0 THEN KOZEPS
1540        LDA K+1
1550        BNE HOPP1
1560 JELZOH DEC KITEVO         ; KITEVO=KITEVO-1
1570        BNE HOPP2          ; IF KITEVO<>0 THEN KULSO
1580        RTS                ; VEGE, VISSZA A HIVOHOZ
1590 KCIM   .WORD 0
```

A fenti program algoritmusa egy kicsit összetettebb az elözönél. Megpróbálunk kulcsot adni a megértéséhez, de igazán csak ugy könnyü megérteni, ha egy rövid (8-10 elemü) lista rendezési példájára az olvasó maga követi papiron vagy akár számitógépen, lépésröl-lépésre végrehajtva a programot. Ezt a könyv példáit tartalmazó lemezen lévö MERGEBAS-TESZT program igen nagymértékben elösegiti. Az alábbiakban utasitásról -utasitásra értelmezzük a programot:

320-330. sor - A már ismert módon átvesszük a hivó SYS utasitás elsö paraméterét, a kezdöelemet ill. annak cimét a KCIM-ben. Megjegyzendö, hogy ez a szubrutin nem az egész tömböt rendezi,hanem csak a megadott elemmel kezdödö, megadott számu elemet. A szubrutin nem végez méretellenörzést, amiböl igen komoly programhibák származhatnak. Szép feladat a 400. sor után ilyen ellenörzés beiktatása,amely BAD SUBSCRIPT ERROR hibát ad a tömb méretének tullépése esetén.

340-400. sor - A második paraméter, a rendezendö elemek számának átvétele és elhelyezése kétbyteos egészként a MERET nevü változóban.

410-450. sor - A NAGYSG-ot 1-re, a kitevöt 0-ra állitjuk.

460-540. sor - Megkeressük azt a 2-es hatványt, amelynek értéke nagyobb vagy egyenlö a mérettel. Ezt ugy végezzük, hogy ciklusban addig szorozzuk a NAGYSG-ot 2-vel, amig kisebb méretnél. A ciklusok számát KITEVO-ben számoljuk.

550-560. sor - Ez a legkülsö ciklus feje. Ebben a ciklusban (amely KITEVO-ször fut le) egy rendezési menet zajlik le. A részlistákat teljesen lerendezzük buborék-eljárással. Az egyes menetekben annyi részre osztjuk a teljes halmazt, amennyi NAGYSG értéke és ezeket a részeket egymástól teljesen függetlenül rendezzük, bár a program szervezésében ez összefonódik. Mivel NAGYSG értékét a ciklus elején rögtön felére csökkentjük, az elsö menetben a listák száma kettönek az a hatványa, ami még éppen kisebb MERET-nél. Ezután minden legkülsö cikluslépésben felére csökken - éppen ezen két utasitás végrehajtásával - NAGYSG értéke, egészen addig, mig az utolsó menetben egy lesz, azaz a teljes tömböt egészben kell lerendezni.

580-630. sor - A K ciklusváltozó a középsö ciklusban, értéke MERET-NAGYSG, lényegében az utolsó részlista elsö eleme és a tömb vége közötti távolságot adja meg.

640-650. sor - A középsö ciklus kezdete. A ciklus egy-egy lépése a buborék-rendezés külsö cikluslépésinek felel meg. Egy-egy cikluslépésben - a belsö ciklus szerint - valamennyi listát rendezi párhuzamosan. Elsö ténykedése a csere-jelzö 0-ra állitása - szerepe ugyanaz, mint a buborék-eljárásnál.

660-670. sor – A belsö ciklusváltozó 0-zása.

680-710. sor – Az összehasonlitási ciklus elsö eleme a teljes lista elsö eleme lesz, ennek cimét KCIM-böl vesszük.

720-840. sor – Itt kezdödik a legbelsö ciklus, ami annyi lépésböl áll, amennyi K értéke, ezzel valamennyi listában megtesszük a soronkövetkezö összehasonlitást. A ciklusban az elsö teendö a két összehasonlitandó bejegyzés hosszának kiemelése a leirókból és a HOSSZ1 és HOSSZ2 munkaváltozókba tárolása.

910-990. sor – Ugyanezt tesszük a leiró második részével, a karakterláncok kezdöcimével is. Ezeket STR1 és STR2, a nullás lapon elhelyezkedö pointerekbe irjuk.

1000. sor – Az Y regisztert nullázzuk az indirekt indexelt cimzéshez. Az Y regiszter igy tehát a stringek elsö (0.) byte-jára mutat.

1010-1100. sor – Ebben a ciklusban történik a karaktersorozatok összehasonlitása. Ha ennek n.-edik lépésében

- a második string-beli karakter kisebb (a töle balra lévök idáig egyenlök voltak!), a második karaktersorozat kisebb, csere szükséges;
- az elsö karaktersorozat aktuális karaktere a kisebb, az elsö string kisebb, nem cserélünk;
- az elsö karakterlánc véget ért, definició szerint kisebb,nem kell cserélni;
- a második karakterlánc ért véget, ö a kisebb, cserélni kell;
- a karakterek egyenlöek, mindkét lánc még tart, folytatjuk az összahasonlitást.

1110-1120. sor – Ez a két utasitás a program végéröl továbbitja az ugrásokat, mivel a cél már távolabb van az ugró utasitástól 128 byte-nál.

1130-1140. sor – Ide kerül a vezérlés csere esetén. Elöször a csere-jelzöt állitjuk 0-tól különbözö értékre.

1150-1220. sor – A karaktersorozatok cseréje helyett mindössze a hárombyte-os leirókat cseréljük ki. Ezt ebben a ciklusban tesszük, egy menetben egy byte-ot. Átmeneti tárolásra az X regisztert használjuk.

1230-1250. sor - Ha nem cserélunk, rögtön, ha igen a csere után lépünk ide. Megnöveljük eggyel a ciklusváltozó értékét.

1260-1370. sor - Mindkét elemre mutató pointer értékét 3 byte-tal, egy elemmel elöbbre állitjuk, vagyis áttérünk a következö részlista következö elemére.

1380-1430. sor - Megvizsgáljuk, hogy elértük-e a belsö ciklus végét, ha nem, folytatjuk BELSO-nél.

1440-1450. sor - Ha nem volt csere, a külsö ciklus végellenörzéséhez ugrunk, mivel ez azt jelenti, hogy a részlisták már rendezettek.

1460-1520. sor - Csökkentjük a középsö ciklus változóját is.

1530-1550. sor - Ha nem csökkent 0-ra, az egyes részrendezések következö lépésére térunk át, azaz a középso cilkus folytatódik.

1560. sor - Csökkentjük a kitevö értékét.

1570. sor - Ha nem 0, visszatérünk a külso ciklus elejére és most feleannyi részlistát fogunk rendezni. Ha 0, már az egész lista rendezett, tehát a 2^0=1 részlistát is lerendeztük.

1580. sor - Visszatérés a BASIC programba.

Ez az eljárás lényegesen jobb eredményt ad, mint az elozo, még nagyobb elemszámok esetén is. Tájékoztatásul: 1000 db, átlagosan 8 hosszuságu véletlenszerüen generált string rendezése kb. 21 másodpercet igényelt, ami nem rossz eredmény.

A programot házi feladatként érdemes továbbfejleszteni. Például a teljes stringek helyett csak adott poziciótól kezdödö adott számu, vagy elöre definiált terminátorig terjedö karakterüket tekinteni rendezési kulcsnak és eszerint teljes rekordokat rendezni. Ennél sokkal egyszerübb feladat lehet az itt adott növekvö rendezés helyett parameterezhetöen csökkenö vagy növekvö rendezést használni. Ugyancsak érdekes feladatot jelenthet az azonos kulcsok közül az elsö kivételével mind elhagyása és a keletkézett tömb tömöritése, valamint a megmaradt elemek számának kimenö paraméterként való visszaadása, stb. Az ilyen feladatok megoldását, persze teszteléssel együtt, amolyan kis **házi vizsgaként** is lehet értékelni. Aki pl. a legutolsó átalakitást meg tudja oldani, nyugodt lehet,elsajátitotta a könyvben tárgyalt ismereteket.

5.6 Nagy feladatok megközelitése

Mig az eddigiekben jobbára csak rövid, kiegészitőként alkalmazott programocskákról, szubrutinokról volt szó, könyvünk utolsó fejezetében szeretnénk néhány jótanácsot adni **utravalóul** azoknak, akik ennél nagyobb fába vágják fejszéjüket és önálló, nagyterjedelmü software-terméket kivánnak létrehozni.

A nagyobb programozási feladatok tulajdonképpen egy hosszabb rendszerfejlesztési tevékenységnek csak kis részét képezik és ennek megfelelöen kell kezelni öket.A rendszerfejlesztési tevékenységnek - anélkül,hogy most szervezési fejtegetésekbe bocsátkoznánk - egy lehetséges felosztás szerint a következö fö részei vannak:

1. Helyzetfelmérés,a rendszerrel szemben támasztott input és output igények,valamint hatékonysági követelmények megismerése.

2. Külsö specifikáció, vagy nagyvonalu rendszerterv készitése, amely dönt a megoldásban felhasználható hardware és software elemekröl, pontosan leirja a kész rendszerrel szemben támasztott követelményeket, a termék elkészülését bizonyitó átvételi teszt mélységéig.

3. Részletes rendszerterv kidolgozása, amely mezö mélységig specifikálja az adatállományokat - adatfeldolgozás esetén - és algoritmus - blokkdiagram vagy egyéb kodifikált algoritmus-leirás - mélységig a programot. Ezáltal programozásra alkalmas szervezési dokumentációként kezelhetö.

4. Programozási tevékenységek, amelyek programtervezésböl, tesztelési terv elkészitéséböl és magából a kódolásból állanak.

5. Programtesztelési munkálatok, amelynek során elöször modulonként, majd a kész modulokat programmá integrálva programonként, végül a teljes rendszerre teljesitjük a tesztelési tervben foglaltakat.

6. A rendszeren lefuttatjuk az átvételi tesztet, hatékonysági vizsgálatot végzünk, javaslatot teszünk kódoptimalizálásra és visszatérünk a 4. ponthoz, vagy befejezettnek nyilvánitjuk a terméket.

7. A szerviztevékenység folyamatos ellátása.

Ezen lépések bármelyikében olyan döntésekre kényszerülhetünk, hogy valamelyik korábbi tevékenységi pontra visszatérve az azóta elkészült dokumentumok és termékek átdol-

gozásával ill. ujra elkészitésével következetesen folytassuk a munkamenetet. **Az olyan – elegendöen nagyméretü – programok, amelyek nem a fenti vagy ahhoz hasonló szervezett módon készülnek, megbizhatatlan kódot eredményeznek**, amelynek utólagos szerviz-ráforditásai messze meghaladják azt a költséget, amelyet a helyesen végzett fejlesztö tevékenység igényelt volna.

A teljes rendszerfejlesztési vertikum véghezvitele a programozási szaktudáson kivül egyéb képzettséget, elsösorban **szervezöi ismereteket** is igényel. Mivel a mikroszámitógépek területén igen gyakori az egyszemélyes, vagy igen kis csoport által végzett munka, különösen fontos, hogy az ilyen körülmények között dolgozó szakemberek bizonyos fokig univerzálisak legyenek. Gyakran lebecsülik a kisgépes fejlesztöi tevékenységet, s mig a nagygépeknél szinte mindig betartják a fenti, vagy ahhoz hasonló munkaszervezés lépcsöfokait és az egyes lépésekben megfelelö képzettségü specialistákat alkalmaznak, addig a kisgépes rendszerek készitését kis munkának tekintik. Ez igen nagy és igen veszélyes tévedés, mivel a mikrogépek általában korlátozott software-ellátottsága és néha nagyságrendekkel kisebb hatékonysági mutatói mellett igen magas felhasználói igényeket megkiváno rendszereket kell megvalósitani. Valószinüleg ennek a lebecsülésnek a következnémye az is, hogy feltünöen sok rossz minöségü programterméket kinálnak szerte a világban személyi számitógépekre.

Ha a nagyobb rendszerek fejlesztésében szükséges programozási feladatokra koncentrálunk, igen fontos már **a tervezési szakaszban**

- a lehetö legkisebb részekre – akár assembly-makró szintig – dekomponálni a feladatot és definiálni az egyes elemek pontos interface-ét. Általában ezen tevékenység közben kiderülnek a negyvonalu terv sulyos hibái és kialakul, mintegy elveszti szabadságfokát a modul algoritmusa.

- Meg kell tervezni a tesztelés pontos tervét modulonként, mivel nagyobb mennyiségü teszteletlen kód átlátása megoldhatatlan nehézségeket okozhat. Nem szabad sajnálni a fáradtságot a modulok számára megfelelö teszt-ágy készitésére, amely hüen szimulálja a többi modul interface-szerinti viselkedését. A tesztelési tervnek pontosan tartalmaznia kell a tesztek várható eredményeit is és ennek ellenörzését sem árt tesztágyra bizni.

A kódolási szakaszban

- nem szabad eltérni a programtervtöl, teljesithetetlenség esetén vissza kell menni a tervezési szakaszhoz.

A tesztelés során

- pontosan ragaszkodni kell a megtervezett tesztelési folyamathoz,

- csak tökéletesen letesztelt modulokat integráljunk - azokból sem tul sokat - nagyobb egységekké. Soha ne spóroljuk meg a teszt-ágy készitését magunk vagy mások által leteszteltnek hitt modulok alkalmazásával!

A teljes munka alatt pontosan **dokumentáljuk** tevékenységünket, bátran használjuk a számitógépet ennek támogatásában! Minél több szervezési, tervezési, programozási, tesztelési dokumentációt örzünk meg, annál könnyebb lesz a hibák felderitése. Ne sajnáljuk a dokumentálásra forditott idöt akkor sem, ha mi magunk vagyunk programunk megrendelöje, szervezöje, tervezöje, kódolója, tesztelöje és végfelhasználója egy személyben! **Semmi sem tud olyan ismeretlen lenni, mint a fél éve általunk irt program!**

Végül, ha jóminöségü terméket akarunk létrehozni, vegyük figyelembe a személyi számitógépekre irt programok néhány speciális **külsö minöségtényezöjét:**

- A képernyö legyen jól szervezett, áttekinthetö, nem tul zsufolt. Kerüljük a buta BASIC programokra jellemzö rollozást.

- A képernyön mindig történjen valami. Bármilyen bonyolult feladatot is old meg programunk, a felhasználó egy perc után feltétlenül - néha már hamarabb - gyanakodni kezd.

- Segitsük a gyakorlatlan felhasználót jól áttekinthetö, érthetö pontokból álló menükkel, könnyen kiválthatò help-funkciókkal.

- Ne gátoljuk a gyakorlott felhasználót mindezen - hosszantartó - elemekkel, adjunk lehetöséget számára rövid, sokatmondó részekböl álló összetett parancsok kiadására.

- Bátran - de mértékletesen - használjuk ki a gép speciális lehetöségeit, grafikát, hangot, joystick-et, fényceruzát, stb. Ugyanakkor adjuk meg a lehetöségét annak is, hogy az ilyen speciális eszközökkel nem rendelkezö felhasználók is használhassák a programot.

A fenti felhasználói kapcsolat-elemek példaszerü megvalósitásával találkozhat az olvasó a **SUPERBASE 64** programban (más kérdés, hogy adatbáziskezelésre mennyire alkalmas).

Végül a mikroszámitógépes rendszerfejlesztési tevékenység talán legfontosabb pontjára szeretnénk felhivni az olvasó figyelmét. Ne próbáljunk minden felmerülö feladatot feltétlenül megoldani Commodore 64 segitségével. Azokról az igényekröl, amelyek, ugy tünik, éppenhogy megoldhatók a géppel, menetközben kiderül, hogy sokkal ágasbogasabbak és ilyenkor már nehéz abbahagyni a munkát. Igy csak lábujjhegyen álló, konnyen összeomló rendszerek készithetök, amelyek akkor mondják fel a szolgálatot, amikor a legkellemetlenebb. **Használjuk a Commodore 64-et arra, amire való** és sok örömünk lesz mind a programok fejlesztésében, mind alkalmazásában.

A FUGGELEK

COMMODORE 64 KODTABLA

DEC	HEXA	ASCII	SCREEN	BASIC	GEPIKOD	DEC	HEXA
0	00		@	SORVEG	BRK	0	00
1	01		A		ORA/IX	1	01
2	02		B			2	02
3	03		C			3	03
4	04		D			4	04
5	05		E		ORA/N	5	05
6	06		F		ASL/N	6	06
7	07		G			7	07
8	08		H		PHP	8	08
9	09		I		ORA/KV	9	09
10	0A		J		ASL/KV	10	0A
11	0B		K			11	0B
12	0C		L			12	0C
13	0D	RETURN	M		ORA/A	13	0D
14	0E		N		ASL/BE	14	0E
15	0F		O			15	0F
16	10		P		BPL/A	16	10
17	11	CRSR LE	Q		ORA/IY	17	11
18	12	RVS ON	R			18	12
19	13	CRSHOME	S			19	13
20	14	DELETE	T			20	14
21	15		U		ORA/NX	21	15
22	16		V		ASL/NX	22	16
23	17		W			23	17
24	18		X		CLC	24	18
25	19		Y		ORA/AY	25	19
26	1A		Z			26	1A
27	1B		[			27	1B
28	1C		£			28	1C
29	1D	CRSJOBB	]		ORA/AX	29	1D
30	1E		↑		ASL/AX	30	1E
31	1F		←			31	1F
32	20				JSR	32	20
33	21	!	!	!	AND/IX	33	21
34	22	"	"	"		34	22
35	23	#	#	#		35	23
36	24	$	$	$	BIT/N	36	24
37	25	%	%	%	AND/Z	37	25
38	26	&	&	&	ROL/Z	38	26
39	27	'	'	'		39	27
40	28	(	(	(	PLP	40	28
41	29	)	)	)	AND/KV	41	29
42	2A	*	*	*	ROL/BE	42	2A
43	2B	+	+	+	ROL/Z	43	2B
44	2C	,	,	,	BIT/A	44	2C
45	2D	-	-	-	AND	45	2D
46	2E	.	.	.	ROL	46	2E
47	2F	/	/	/		47	2F
48	30	0	0	0	BMI	48	30
49	31	1	1	1	AND/IY	49	31

DEC	HEX	[illegible]	[illegible]	[illegible]	[illegible]	DEC	HEX
50	32	2	2	2		50	32
51	33	3	3	3		51	33
52	34	4	4	4		52	34
53	35	5	5	5	AND/NX	53	35
54	36	6	6	6	ROL/NX	54	36
55	37	7	7	7		55	37
56	38	8	8	8	SEC	56	38
57	39	9	9	9	AND/AY	57	39
58	3A	:	:	:		58	3A
59	3B	;	;	;		59	3B
60	3C	<	<	<		60	3C
61	3D	=	=	=	AND/AX	61	3D
62	3E	>	>	>	ROL/AX	62	3E
63	3F	?	?	?		63	3F
64	40	@		@	RTI	64	40
65	41	A		A	EOR/IX	65	41
66	42	B		B		66	42
67	43	C		C		67	43
68	44	D		D		68	44
69	45	E		E	EOR/N	69	45
70	46	F		F	LSR/N	70	46
71	47	G		G		71	47
72	48	H		H	PHA	72	48
73	49	I		I	EOR/KV	73	49
74	4A	J		J	LSR/BE	74	4A
75	4B	K		K		75	4B
76	4C	L		L	JMP/A	76	4C
77	4D	M		M	EOR/A	77	4D
78	4E	N		N	LSR/A	78	4E
79	4F	O		O		79	4F
80	50	P		P	BVC	80	50
81	51	Q		Q	EOR/IY	81	51
82	52	R		R		82	52
83	53	S		S		83	53
84	54	T		T		84	54
85	55	U		U	EOR/NX	85	55
86	56	V		V	LSR/NX	86	56
87	57	W		W		87	57
88	58	X		X	CLI	88	58
89	59	Y		Y	EOR/AY	89	59
90	5A	Z		Z		90	5A
91	5B					91	5B
92	5C					92	5C
93	5D				EOR/AX	93	5D
94	5E				LSR/AX	94	5E
95	5F					95	5F
96	60				RTS	96	60
97	61		▌		ADC/IX	97	61
98	62		▄			98	62
99	63		▔			99	63

DEC	HEXA	ASCII KÓD	SCREEN	BASIC KÓD	GÉPI KÓD	DEC	HEXA
100	64		▁			100	64
101	65		▏		ADC/N	101	65
102	66		▒		ROR/N	102	66
103	67		▕			103	67
104	68		▒		PLA	104	68
105	69		◤		ADC/KV	105	69
106	6A		▕		ROR/BE	106	6A
107	6B		├			107	6B
108	6C		▗		JMP/I	108	6C
109	6D		└		ADC/A	109	6D
110	6E		┐		ROR/A	110	6E
111	6F		▂			111	6F
112	70		┌		BVS	112	70
113	71		┴		ADC/IY	113	71
114	72		┬			114	72
115	73		┤			115	73
116	74		▏			116	74
117	75		▌		ADC/NX	117	75
118	76		▐		ROR/NX	118	76
119	77		▔			119	77
120	78		▀		SEI	120	78
121	79		▄		ADC/AY	121	79
122	7A		┘			122	7A
123	7B		▖			123	7B
124	7C		▝			124	7C
125	7D		┘		ADC/AX	125	7D
126	7E		▘		ROR/AX	126	7E
127	7F		▚			127	7F
128	80		▣	END		128	80
129	81		▣	FOR	STA/IX	129	81
130	82		▣	NEXT		130	82
131	83		▣	DATA		131	83
132	84		▣	INPUT#	STY/N	132	84
133	85		▣	INPUT	STA/N	133	85
134	86		▣	DIM	STX/N	134	86
135	87		▣	READ		135	87
136	88		▣	LET	DEY	136	88
137	89		▣	GOTO		137	89
138	8A		▣	RUN	TXA	138	8A
139	8B		▣	IF		139	8B
140	8C		▣	RESTORE	STY/A	140	8C
141	8D		▣	GOSUB	STA/A	141	8D
142	8E		▣	RETURN	STX/A	142	8E
143	8F		▣	REM		143	8F
144	90		▣	STOP	BCC	144	90
145	91	CRSRFEL	▣	ON	STA/IY	145	91
146	92	RVS OFF	▣	WAIT		146	92
147	93	CLR	▣	LOAD		147	93
148	94	INST	▣	SAVE	STY/NX	148	94
149	95		▣	VERIFY	STA/NX	149	95

DEC	HEX	ASCII	SCREEN	BASIC	6502	DEC	HEX
150	96			DEF	STX/NY	150	96
151	97			POKE		151	97
152	98			PRINT#	TYA	152	98
153	99			PRINT	STA/AY	153	99
154	9A			CONT	TXS	154	9A
155	9B			LIST		155	9B
156	9C			CLR		156	9C
157	9D	CRSRBAL		CMD	STA/AX	157	9D
158	9E			SYS		158	9E
159	9F			OPEN		159	9F
160	A0			CLOSE	LDY/KV	160	A0
161	A1			GET	LDA/IX	161	A1
162	A2			NEW	LDX/KV	162	A2
163	A3			TAB(		163	A3
164	A4			TO	LDY/N	164	A4
165	A5			FN	LDA/N	165	A5
166	A6			SPC(	LDX/N	166	A6
167	A7			THEN		167	A7
168	A8			NOT	TAY	168	A8
169	A9			STEP	LDA/KV	169	A9
170	AA			+	TAX	170	AA
171	AB			−		171	AB
172	AC			*	LDY/A	172	AC
173	AD			/	LDA/A	173	AD
174	AE				LDX/A	174	AE
175	AF			AND		175	AF
176	B0			OR	BCS	176	B0
177	B1				LDA/IY	177	B1
178	B2			=		178	B2
179	B3					179	B3
180	B4			SGN	LDY/NX	180	B4
181	B5			INT	LDA/NX	181	B5
182	B6			ABS	LDX/NX	182	B6
183	B7			USR		183	B7
184	B8			FRE	CLV	184	B8
185	B9			POS	LDA/AY	185	B9
186	BA			SQR	TSX	186	BA
187	BB			RND		187	BB
188	BC			LOG	LDY/AX	188	BC
189	BD			EXP	LDA/AX	189	BD
190	BE			COS	LDX/AY	190	BE
191	BF			SIN		191	BF
192	C0			TAN	CPY/KV	192	C0
193	C1			ATN	CMP/IX	193	C1
194	C2			PEEK		194	C2
195	C3			LEN		195	C3
196	C4			STR$	CPY/N	196	C4
197	C5			VAL	CMP/N	197	C5
198	C6			ASC	DEC/N	198	C6
199	C7			CHR$		199	C7

DEC	HEXA	ASCII	SCREEN	BASIC	GEPIKOD	DEC	HEXA
200	C8			LEFT$	INY	200	C8
201	C9			RIGHT$	CMP/KV	201	C9
202	CA			MID$	DEX	202	CA
203	CB					203	CB
204	CC				CPY/A	204	CC
205	CD				CMP/A	205	CD
206	CE				DEC/A	206	CE
207	CF					207	CF
208	D0				BNE	208	D0
209	D1				CMP/IY	209	D1
210	D2					210	D2
211	D3					211	D3
212	D4					212	D4
213	D5				CMP/NX	213	D5
214	D6				DEC/NX	214	D6
215	D7					215	D7
216	D8				CLD	216	D8
217	D9				CMP/AY	217	D9
218	DA					218	DA
219	DB					219	DB
220	DC					220	DC
221	DD				CMP/AX	221	DD
222	DE				DEC/AX	222	DE
223	DF					223	DF
224	E0				CPX/KV	224	E0
225	E1				SBC/IX	225	E1
226	E2					226	E2
227	E3					227	E3
228	E4				CPX/N	228	E4
229	E5				SBC/N	229	E5
230	E6				INC/N	230	E6
231	E7					231	E7
232	E8				INX	232	E8
233	E9				SBC/KV	233	E9
234	EA				NOP	234	EA
235	EB					235	EB
236	EC				CPX/A	236	EC
237	ED				SBC/A	237	ED
238	EE				INC/A	238	EE
239	EF					239	EF
240	F0				BEQ	240	F0
241	F1				SBC/IY	241	F1
242	F2					242	F2
243	F3					243	F3
244	F4					244	F4
245	F5				SBC/NX	245	F5
246	F6				INC/NX	246	F6
247	F7					247	F7
248	F8				SED	248	F8
249	F9				SBC/AY	249	F9

DEC	HEX	[illegible]	[illegible]	[illegible]	[illegible]	DEC	HEX
250	FA					250	FA
251	FB					251	FB
252	FC					252	FC
253	FD				SBC/AX	253	FD
254	FE				INC/AX	254	FE
255	FF					255	FF

B. függelék

A MOS 6510 utasitáskészlete

1. Cimzési módok szerint rendezve:

Abszolut: nn

ADC/AND/ASL/BIT/CMP/CPX/CPY/DEC/EOR/INC/JMP/JSR/LDA/
LDX/LDY/LSR/ORA/ROL/ROR/SBC/STA/STX/STY

Közvetlen: #n

ADC/AND/CMP/CPX/CPY/EOR/LDA/LDX/LDY/ORA/SBC

Nullás lapu: ∅n

ADC/AND/ASL/BIT/CMP/CPX/CPY/DEC/EOR/INC/LDA/LDX/LDY/
LSR/ORA/ROL/ROR/SBC/STA/STX/STY

Beleértett: R

ASL/BRK/CLC/CLD/CLI/CLV/DEX/DEY/INX/INY/LSR/NOP/PHA/
PHP/PLA/PLP/ROL/ROR/RTI/RTS/SEC/SED/SEI/TAX/TAY/TSX/
TXA/TXS/TYA

Relativ: +n

BCC/BCS/BEQ/BMI/BNE/BPL/BVC/BVS

.Indirekt: (nn)

JMP

Abszolut indexelt (X): nn,X

ADC/AND/ASL/CMP/DEC/EOR/INC/LDA/LDY/LSR/ORA/ROL/ROR/
SBC/STA

Abszolut indexelt (Y): nn,Y

ADC/AND/CMP/EOR/LDA/LDX/ORA/SBC/STA

Nullás lapu indexelt (X): ∅n,X

ADC/AND/ASL/CMP/DEC/EOR/INC/LDA/LDY/LSR/ORA/ROL/ROR/
SBC/STA/STY

Nullás lapu indexelt (Y): ∅n,Y

LDX/STX

Indexelt indirekt: (∅n,X)

ADC/AND/CMP/EOR/LDA/ORA/SBC/STA

Indirekt indexelt: (∅n),Y

ADC/AND/CMP/EOR/LDA/ORA/SBC/STA

2. A jelzökre gyakorolt hatás szerint rendezve:

Eredménytöl függöen állitja:

N,Z

ADC/AND/ASL/BIT/CMP/CPX/CPY/DEX/DEY/EOR/INC/INX/INY/LDA/LDX/LDY/LSR/ORA/PLA/PLP/ROL/ROR/RTI/SBC/TAX/TAY/TYA/TSX/TXA

C

ADC/ASL/CMP/CPX/CPY/LSR/PLP/ROL/ROR/SBC

V

ADC/BIT/SBC

A jelzöt törli:

V

CLV

D

CLD

C

CLC

I

CLI

N

LSR

A jelzőt magasra állitja:

D

SED

C

SEC

I

BRK/SEI

KERNAL szubrutinok

ACPTR – adatbyte input a soros buszról
Belépési pont: $FFA5
Paraméterek: A – output/ az olvasott byte
Elökészitö rutinok: TALK, TKSA
Hivása:

```
JSR ACPTR
STA BYTE
```

CHKIN – input csatorna elökészitése olvasásra
Belépési pont: $FFC6
Paraméterek: X – input/ csatornaszám /filesorszám/
Elökészitö rutinok: OPEN
Hivása:

```
LDX #FILE
JSR CHKIN
```

CHKOUT – output csatorna elökészitése olvasásra
Belépési pont: $FFC9
Paraméterek: X – input/ csatornaszám /filesorszám/
Elökészitö rutinok: OPEN
Hivása:

```
LDX #FILE
JSR CHKOUT
```

CHRIN – karakter olvasása az input csatornáról, ha nem elözte meg CHKIN, akkor a billentyüzetröl a képernyöszerkesztö segitségével
Belépési pont: $FFCF
Paraméterek: A – output/ beolvasott byte
Elökészitö rutinok: OPEN, CHKIN
Hivása:

```
JSR CHRIN
STA BYTE
```

CHROUT – karakter küldése az output csatornára, ha nem elözte meg CHKIN, akkor a képernyö aktuális poziciójára
Belépési pont: $FFD2
Paraméterek: A – input/ kiirandó byte
Elökészitö rutinok: OPEN, CHKOUT
Hivása:

```
LDA BYTE
JSR CHRIN
```

CIOUT - +karakter küldése a soros buszra
Belépési pont: $FFA8
Paraméterek: A - input/ kiirandó byte
Elökészitö rutinok: LISTEN, SECOND
Hivása:

```
        LDA BYTE
        JSR CIOUT
```

CINT - a képernyöszerkesztö és a VIC processzor inicia-lizálása
Belépési pont: $FF81
Paraméterek: -
Elökészitö rutinok: -
Hivása:

```
        JSR CINT
```

CLALL - valamennyi csatorna és file lezárása
Belépési pont: $FFE7
Paraméterek: -
Elökészitö rutinok: -
Hivása:

```
        JSR CLALL
```

CLOSE - file lezárása
Belépési pont: $FFC3
Paraméterek: A - input/ filesorszám
Elökészitö rutinok: -
Hivása:

```
        LDX #FILE
        JSR CLOSE
```

CLRCHN - input és output csatorna lezárása és a csatornák eredeti (default) értékének visszaállitása
Belépési pont: $FFCC
Paraméterek: -
Elökészitö rutinok: -
Hivása:

```
        JSR CLRCHN
```

GETIN - karakter olvasása a billentyüzet pufferból
Belépési pont: $FFE4
Paraméterek: A - output/ beolvasott byte vagy 0, ha a puffer üres
Elökészitö rutinok: OPEN, CHKIN
Hivása:

```
CIKLUS JSR CHRIN
       CMP #0
       BEQ CIKLUS
```

IOBASE - input es output eszközök memóriafoglalása
Belépési pont: $FFF3
Paraméterek: X - output/ a memóriacim alsó byte-ja
Y - output/ a memóriacim felsö byte-ja

Elökészitö rutinok: -
Hivása:

```
        JSR IOBASE
```

IOINIT - az input/output eszközök inicializálása
Belépési pont: $FF84
Paraméterek: -
Elökészitö rutinok: -
Hivása:

```
        JSR IOINIT
```

LISTEN - az input/output eszközt hallgató állapotba hozza
Belépési pont: $FFB1
Paraméterek: A - input/ egységszám
Elökészitö rutinok: -
Hivása:

```
        LDA #EGYSEG
        JSR LISTEN
```

LOAD - program-file töltése/ellenörzése a memóriába
Belépési pont: $FFD5
Paraméterek: A - input/ 0=LOAD, 1=VERIFY
ha az eszközt 0 másodlagos cimmel
nyitottuk meg
X - input/ memóriacim alsó byte-ja
Y - input/ memóriacim felsö byte-ja
Elökészitö rutinok: -
Hivása:

```
        LDA #0
        JSR LOAD
```

MEMBOT - beállitja a BASIC ROM kezdöcimét
Belépési pont: $FF9C
Paraméterek: X - input/ memóriacim alsó byte-ja
Y - input/ memóriacim felsö byte-ja
Az alapfeltételezés szerinti kezdöérték $800
Elökészitö rutinok: -
Hivása:

```
        JSR MEMBOT
```

MEMTOP - beállitja a BASIC ROM végcimet
Belépési pont: $FF99
Paraméterek: X - input/ memóriacim alsó byte-ja
Y - input/ memóriacim felsö byte-ja
Elökészitö rutinok: -
Hivása:

```
        JSR MEMTOP
```

OPEN - megnyit egy file-t
Belépési pont: $FFC0
Paraméterek: -
Elökészitö rutinok: SETLFS, SETNAM
Hivása:

```
        JSR OPEN
```

PLOT C=0- beállitja a cursor pozicióját C=1 lekérdezi

Belépési pont: $FFF0

Paraméterek: X - input/ sor (0-24)

Y - input oszlop (0-39)

Elökészitö rutinok: SETLFS, SETNAM

Hivása:

```
LDX #SOR
LDY #OSZLOP
JSR PLOT
```

RAMTES - memóriateszt, majd a memóriában elhelyezett rendszerpointerek és munkaterületek inicializálása

Belépési pont: $FF87

Paraméterek: -

Elökészitö rutinok: -

Hivása:

```
JSR RAMTES
```

RDTIM - kiolvassa a rendszer-órát

Belépési pont: $FFDE

Paraméterek: Y - output/ óra alsó byte-ja

X - output/ óra középsö byte-ja

A - output/ óra felsö byte-ja

Elökészitö rutinok: -

Hivása:

```
JSR RDTIM
STY ORA
STX ORA+1
STA ORA+2
```

READST - kiolvassa az inpu/output állapotszót

Belépési pont: $FFB7

Paraméterek: A - output/ állapotszó

Elökészitö rutinok: -

Hivása:

```
JSR READST
```

RESTOR - visszaállitja eredeti értékeire a rendszervektorokat

Belépési pont: $FF8A

Paraméterek: -

Elökészitö rutinok: -

Hivása:

```
JSR RESTOR
```

SAVE - program-file kimentése a memóriából

Belépési pont: $FFD8

Paraméterek: A - input/ egy nullás lapon lévö cim, amely a program kezdetére mutat

X - input/ a végcim alsó byte-ja

Y - input/ a végcim felsö byte-ja

Elökészitö rutinok: SETLFS,SETNAM

Hívása:

```
        LDA ELEJE
        LDX #VEGE<
        LDY #VEGE>
        JSR SAVE
```

SCNKEY – az eppen megnyomott billentyűnek megfelelö ASCII kódot a billentyüzet pufferébe továbbitja

Belépési pont: $FF9F

Paraméterek: -

Elökészitö rutinok: IOINIT

Hívása:

```
        JSR SCNKEY
```

SCREEN – a képernyö méretét adja vissza a nagyobb Commodore gépeken is futó programokkal való kompatibilitás miatt

Belépési pont: $FFED

Paraméterek: X - output/oszlopok száma
Y - output/sorok száma

Elökészitö rutinok: -

Hívása:

```
        JSR SCREEN
        STX MAXO
        STY MAXS
```

SECOND – másodlagos cim küldése a hallgató állapotu eszköz felé

Belépési pont: $FF93

Paraméterek: A - input/ másodlagos cim

Elökészitö rutinok: LISTEN

Hívása:

```
        LDA #MASODL
        JSR SECOND
```

SETLFS – file-paraméterek beállitása

Belépési pont: $FFBA

Paraméterek: X - input/ filesorszám
Y - input/ másodlagos cim, ha nincs $FF
A - input/ egységszám

Elökészitö rutinok: -

Hívása:

```
        LDX #FILE
        LDY #$FF
        LDA #DEV
        JSR SETLFS
```

SETMSG – üzenetek kiirását engedélyezi illetve tiltja le. Az akkumulátor legmagasabb helyiértékü bitje a hibaüzenetekre (pl. FILE NOT FOUND), a következö bit a vezérlö üzenetekre (pl. PRESS PLAY ON CASETTE) vonatkozik. Ritkán használatos.

Belépési pont: $FF90

Paraméterek: A - input / 7. bit - hibaüzenet
6. bit - vezérlö üzenet

Elökészitö rutinok: -

Hívása:

```
        LDA #$80     ; hibaüzenetek engedélyezése
        JSR SETMSG
```

```
SETNAM   -   file-név beállitása
             Belépési pont: $FFBD
             Paraméterek: X - input/ a név cimének alsó  byte-ja
                          Y - input/ a név cimének felsö byte-ja
                          A - input/ a név hossza
             Elökészitö rutinok: -
             Hivása:
                     LDX #NEV<
                     LDY #NEV>
                     LDA #NEVV-NEV
                     JSR SETNAM

SETTIM   -   a rendszeróra beállitása
             Belépési pont: $FFDB
             Paraméterek: X - input/ az idö középsö byte-ja
                          Y - input/ az idö alsó byte-ja
                          A - input/ az idö felsö byte-ja
             Elökészitö rutinok: -
             Hivása:
                     LDY ORA
                     LDX ORA+1
                     LDA ORA+2
                     JSR SETTIM

SETTMO   -   az IEEE buszon a timeout jelzö  állitása/törlése
             (Ha egy CBM szabvány szerinti IEEE illesztö egy-
             séget használunk.  Az itthon kaphatók  általában
             nem ilyenek!)
             Belépési pont: $FFA2
             Paraméterek: A - input/ a 7. bit magas - törlés
                                            alacsony - beállitás

             Elökészitö rutinok: -
             Hivása:
                     LDA #1  ;TORLES
                     JSR SETTMO

STOP     -   megvizsgálja,  hogy  meg volt-e nyomva a STOP bil-
             lentyü a legutóbbi  UDTIM  hivás alatt,  ha  igen,
             alapállapotra állitja az input és output csatornát
             Belépési pont: $FFE1
             Paraméterek: Z - output/ magas -> volt STOP
             Elökészitö rutinok: -
             Hivása:
                     JSR UDTIM
                     JSR STOP
                     BNE MEGSZ

TALK     -   az input/output eszközt beszélö állapotba hozza
             Belépési pont: $FFB4
             Paraméterek: A - input/ egységszám
             Elökészitö rutinok: -
             Hivása:
                     LDA #EGYSEG
                     JSR TALK
```

TKSA - másodlagos cim küldése a beszélö állapotu eszköz felé
Belépési pont: $FF96
Paraméterek: A - input/ másodlagos cim
Elökészitö rutinok: TALK
Hivása:

```
LDA #MASODL
JSR TKSA
```

UDTIM - aktualizálja - egy egységgel növeli - a rendszerórát, csak akkor kell hivni, ha a megszakitásokat magunk kezeljük
Belépési pont: $FFEA
Paraméterek: -
Elökészitö rutinok: -
Hivása:

```
JSR UDTIM
```

UNLSN - valamennyi hallgató állapotu input/output eszközt leállitja
Belépési pont: $FFAE
Paraméterek: -
Elökészitö rutinok: -
Hivása:

```
JSR UNLSN
```

UNTLK - valamennyi beszélö állapotu input/output eszközt leállitja
Belépési pont: $FFAB
Paraméterek: -
Elökészitö rutinok: -
Hivása:

```
JSR UNTLK
```

VECTOR - a memóriában lévö rendszer-vektorok kezdöcimének lekérdezése/beállitása
Belépési pont: $FF8D

```
Paraméterek: C - input/ magas -> lekérdezés
             X - output/ a cim alsó byte-ja
             Y - output/ a cim felsö byte-ja
        vagy C - input/ alacsony -> beállitás
             X - input/ a cim alsó  byte-ja
             Y - input/ a cim felsö byte-ja
```

Elökészitö rutinok: -
Hivása:

```
          LDX #VEC<
          LDY #VEC>
          CLC
          JSR VECTOR
     vagy
          SEC
          JSR VECTOR
          STX VECCIM
          STY VECCIM+1
```

Ajánlott és részben felhasznált irodalom:

1. Ury L.: Commodore 64 BASIC felhasználó kézikönyv
(LSI ATSZ, BP. 1984)

2. Commodore 64 Programmer's Reference Guide
Commodore Business Machines, Inc. 1982 angol nyelven

3. Angerhausen, M.; Becker, A. et al.: The Anatomy of the Commodore 64
Abacus Software, Grand Rapids 1984 angol nyelven
vagy
Angerhausen, M.; Becker, A. et al.: Commodore 64 Intern
Data Becker, Düsseldorf 1983
(fentivel megegyező kiadás) német nyelven

4. French, D.: Inside the Commodore 64
French Silk, Minneapolis 1984 angol nyelven

5. Greeshields, M.: 40 Best Machine Code Routines For Commodore 64
Duckworth, London, 1984 angol nyelven

6. Stephenson, A.P. and Stephenson, D.J.: Advanced Machine Code Programming for the Commodore 64
Granada, London 1984 angol nyelven

7. Butterfield, J.: Machine Language for the Commodore 64 and Other Commodore Computers
Brady Communications Company, Inc. Bowie 1984
angol nyelven

8. English, L.: Das Maschinensprache Buch zum Commodore 64
Data Becker, Düsseldorf 1984 német nyelven

9. Lawrence, D. and England, M.: Commodore 64 Machine Code Master
Sunshine Books, London 1983 angol nyelven

Készült: MÁTRAINVEST GT Nyomda
Felelős kiadó: Kovács Magda
Felelős vezető: dr. Csák Máté
Terjedelem: 18,5 B/5 ív
Engedély szám: 53629

COMMODORE-64 oktatócsomag

Az LSI ATSZ legújabb kínálata a **Commodore 64 oktatócsomag**, ami mind a Commodore 64 személyi számítógép napi használatához, mind programozásának elsajátításához szükséges ismereteket tartalmazza.

Az oktatócsomag az alábbiakat tartalmazza:

dr. Úry László: *Commodore-64 BASIC felhasználói kézikönyv*

A kézikönyv második bővített, kiadása a C-64 BASIC-jén túlmenően részletesen ismerteti a SIMONS' BASIC, a SUPERGRAPHIC utasításkészletét, a BASIC 4.0-ás bővítést. Tartalmazza a C-64 grafikájának, illetve hanggenerátorának használatához szükséges ismereteket. A könyv ismerteti a C-64-hez hazánkban vásárolható valamennyi típusú lemezegység használatát, beleértve az IEEE-illesztésű lemezegységekét is.

Erdős Iván: *A Commodore-64 assembly nyelvű programozása*

A könyv részletesen ismerteti az M6510-es utasításkészleteét. Részletesen tárgyalja a Commodore-64 assembly programozását segítő szoftver eszközöket, s bőséges példaanyaggal segíti a témával most ismerkedőket. Ismerteti a C-64 ROM-jának legfontosabb részeit.

dr. Úry László: *Commodore-64 Információs kártya*

Az információs kártya a Commodore-64 használatához szükséges ismereteket foglalja össze táblázatos formában, beleértve a BASIC szintaxisát, a BASIC és a DOS hibaüzeneteket és az M6510-es processzor teljes utasításkészletét.

Commodore oktatólemez

Az oktatólemez az előző két könyv példaanyagán túl további kidolgozott assembly és BASIC programozási példákat tartalmaz.

Az oktatócsomag ára: 3500 Ft.